LEVEL II

(2)

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD A090313

DTIC
ELECTE
OCT 15 1980
C

THESIS

THE DESIGN AND IMPLEMENTATION
OF THE MEMORY MANAGER FOR A
SECURE ARCHIVAL STORAGE SYSTEM

by

Edmund E. Moore
Alan V. Gary

June 1980

Thesis Advisor: L. A. Cox, Jr.

Approved for public release: distribution unlimited

DDC FILE COPY

807

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A090 313	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Design and Implementation of the Memory Manager for a Secure Archival Storage System		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis: June, 1980
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Alan V. Gary Edmund E. Moore		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1980 13. NUMBER OF PAGES 165
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release: distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Memory Manager, Security Kernel, Operating System Security, Distributed Process, Segmentation, Process Switching, File System, Non-Distributed Process, Protection Domain, Aliasing

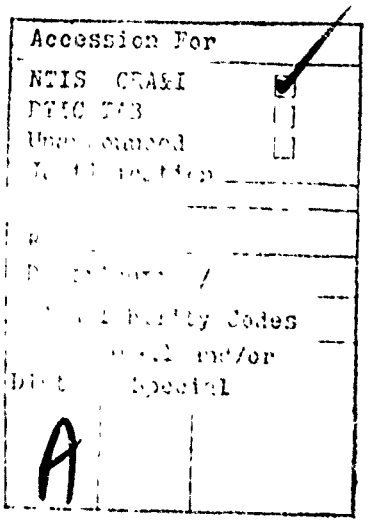
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis presents a detailed design and implementation of a memory manager for a kernel technology based secure archival storage system (SASS). The memory manager is a part of the non-distributed portion of the Security Kernel, and is solely responsible for the proper management of both the main memory (random access) and the secondary storage (direct access) of the system. The memory manager is designed for implementation on the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601
(Page 1)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ZILOG Z8000 microprocessor in a multi-processor environment. The loop free design structure, based upon levels of abstraction, and a segment aliasing scheme for information confinement are essential elements of the overall system security provided by the SASS.



Approved for public release; distribution unlimited.

The Design and Implementation of the Memory Manager for a Secure Archival Storage System

by

Edmund E. Moore
Lieutenant Commander, United States Navy
B.S., United States Naval Academy, 1970

Alan V. Gary
Lieutenant, United States Navy
B.A., University of Louisville, 1974

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1980

Authors Edmund E Moore

Alan V Gary

Approved by: J. R. Cox Jr.
Thesis Advisor

Roger R. Schell
Second Reader

Chairman, Department of Computer Science

D. A. Schrady
Dean of Information and Policy Sciences

ABSTRACT

This thesis presents a detailed design and implementation of a memory manager for a kernel technology based secure archival storage system (SASS). The memory manager is part of the non-distributed portion of the Security Kernel, and is solely responsible for the proper management of both the main memory (random access) and the secondary storage (direct access) of the system. The memory manager is designed for implementation on the ZILOG Z8000 microprocessor in a multi-processor environment. The loop free design structure, based upon levels of abstraction, and a segment aliasing scheme for information confinement are essential elements of the overall system security provided by the SASS.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

This research is sponsored in part by the Office of Naval Research Project Number NR 337-025, monitored by Mr. Joel Trimble.

The support and assistance of Lt.Col Roger Schell, Professor Lyle Cox, Lcdr. Steve Reitz, and lab technicians Mr. Bob McDonnell and Mr. Mike Williams were greatly appreciated. Special thanks go to Barbara Gary for her undivided support, assistance, and patience.

I. INTRODUCTION

This thesis addresses the design and partial implementation of a memory manager for a member of the family of secure, distributed, multi-microprocessor operating systems designed by Richardson and O'Connell [1]. The memory manager is responsible for the secure management of the main memory and secondary storage. The memory manager design was approached and conducted with distributed processing, multi-processing, configuration independence, ease of change, and internal computer security as primary goals. The problems faced in the design were:

1) Developing a process which would securely manage files in a multi-processor environment.
2) Ensuring that if secondary storage was inadvertantly damaged, it could usually be recreated.
3) Minimizing secondary storage accesses.
4) Proper parameter passing during interprocess communication.
5) Developing a process with a loop-free structure which is configuration independent.
6) Designing databases which optimize the memory management functions.

The proper design and implementation of a memory management process is vital because it serves as the

interface between the physical storage of files in a storage system and the logical hierarchical file structure as viewed by the user (viz., the file system supervisor design by Parks [2]). If the memory manager process does not function properly, the security of that system cannot be guaranteed.

The secure family of operating systems designed by Richardson and O'Connell is composed of two primary modules, the supervisor and the security kernel. A subset of that system was utilized in the design of the Secure Archival Storage System (SASS). The design of the SASS supervisor was addressed by Parks [2], while the security kernel was addressed concurrently by Coleman [3]. The SASS security kernel design is composed of two parts, the distributed kernel and the non-distributed kernel. The design of the distributed kernel was conducted by Coleman [3], and processor management was implemented by Reitz [4]. This thesis presents the design and implementation of the non-distributed kernel. In the SASS design, the non-distributed kernel consists solely of the memory manager.

The design of the memory manager and its data bases was completed. The initial code was written in PLZ/SYS, but could not be compiled due to the lack of a PLZ/SYS compiler. A thread of the high level code was selected, hand compiled into PLZ/ASM, and run on the Z8000 developmental module.

The PIM/ASM thread listing is presented as a computer program appended to this thesis.

A. BACKGROUND

Operating systems were initally developed during an era when hardware was a scarce and expensive resource, while software was relatively inexpensive. The initial system design technique was to begin with the hardware configuration and to build the operating system upon it. The "bottom up" design technique was practical, but it made the operating system extremely hardware dependent. Hardware configuration changes would often force a major software redesign, but as long as hardware costs were dominant, software modification was the logical alternative. As the functions required of the operating system increased, new procedures were haphazardly added to the operating system, often introducing new problems. Maintenance and debugging of the operating system became extremely cumbersome and time consuming.

The increased usage of computers in such fields as finance and sensitive information handling uncovered a serious problem with most operating systems. Information stored within a computer system was generally quite accessible to anyone who had a working knowledge of operating system design and structure, regardless of any

ad-hoc attempts to provide internal computer security. Data stored in information systems, with security added in, could not be certified as being totally secure[14].

Recent technological developments have reversed the economics of the computer design environment. Microprocessors have become abundant, powerful, and inexpensive. The relative cost of software, on the otherhand, has steadily increased until it now dominates the overall cost of a computer system. This reversal has two basic implications. First, software must be treated as the expensive commodity. Software developed should therefore be logical, easy to read, relatively maintenance free, and easy to debug. Second, more powerful hardware can be used to perform functions previously performed with software, and thus hardware (multiprocessors) can be utilized to achieve overall system speed goals.

The SASS was developed utilizing a "top down" design technique, with information security as a primary design issue. Security was designed into the system based upon the security kernel concept [5]. The security kernel provides a secure environment by ensuring that just one element of the system (the security kernel) is sufficient to provide the internal system security. All accesses of data stored within the computer system must be validated by the security kernel.

B. BASIC CONCEPTS/DEFINITIONS

1. Process

Organick [6] defines a process as a set of related procedures and data undergoing execution and manipulation, respectively, by one of possibly several processors of a computer. The process is a logical rather than a physical entity, and can be viewed as a set of related procedures and data (referred to as the process' address space) and a point of execution within that address space. Each process may have associated with it such logical attributes as a security class authorization and a unique identifier. In order to execute, the process must be mapped onto (bound to) a physical processor within the computer system.

A process may exist in one of three states: blocked, ready, or running. When in a blocked state, the process must wait for the occurrence of some event before execution can continue (for example, an access of secondary storage). When the event for which a blocked process is waiting occurs, the process is placed into the ready state which indicates that the process can run when a processor is available to be assigned to it. The process is in the running state when it is executing on a processor.

2. Process Switching

When a process is blocked, the physical processor upon which it is scheduled is idle. For efficiency reasons, it makes sense to freeze that process, save the execution point (program status registers, program counter, execution stack) and the address space, and then schedule another process to run on that processor. This is referred to as process switching (or multiprogramming), and is an important aspect of a distributed operating system. The overall system, such as SASS, can be viewed as a set of cooperating processes that interact to perform the intended functions.

Efficient process switching can only be achieved with the support of some hardware switching mechanism that will unload the blocked process' address space, and load the address space of the scheduled process. Some systems have a DBR (descriptor base register) which is used to point to a list of multiple address spaces (one per process) which exists in memory. Thus to change an address space, the DBR need only be changed. The SASS utilizes a Z-8000 supporting hardware device entitled a Memory Management Unit (MMU) to allow efficient process switching. The MMU consists of a set of registers (64 or 128 in the SASS design) which contain the process' address space. Thus process switching would involve the switching of control to another hardware MMU (if a hardware MMU were available for each process), or

alternately loading a software MMU image (which is always kept current) into the MMU whenever a process switch is required. The SASS currently maintains a software MMU image for each process.

3. Protection Domains

A user's process executing on a computer system has an address space which includes the user provided procedures and data, and also those portions of the distributed operating system which are required to support execution of his program. To maintain system integrity and security, it becomes mandatory to protect the operating system from being altered or manipulated by the user's procedures. To achieve this, the process' address space is divided into a set of hierarchical domains which ensure that the segments of the operating system are protected from the user. Since the top down design of the operating system provides a strict hierarchal structure, the domains of the operating system are also hierarchical in structure (viz., are protection rings). In the design of the secure operating system family, three domains were defined: the user, the supervisor, and the kernel.

Operating system segments which manage the actual shared physical resources reside in the kernel. The kernel is the most privileged domain of the address space. It can be envisioned as a mini-operating system that does all the

resource management. The security kernel segments (executable) can only be accessed within the kernel. Global (system wide) data bases are restricted to access by only the security kernel to prevent the possibility of an unauthorized inter-process leakage of information [7].

The supervisor domain resides between the most privileged kernel domain and the least privileged user domain. The supervisor contains those segments of the operating system which are required to provide such common services as creating a hierarchical file system. The supervisor deals with the logical entities (segments) as viewed by the user, and manages these segments by calls to the kernel. To preserve the integrity of the file system, the user is placed in the least privileged domain, and can communicate directly with the supervisor only.

Multiple protection domains may be implemented via either a hardware and/or a software ring structure. A hardware implementation is more efficient, however the VLSI microprocessors currently being manufactured provide for only two protection domains. The present design of the SASS requires two domains, separating the supervisor and the security kernel. The Z8000 microprocessor provides the SASS with the hardware ring structure by providing two execution modes, the system mode and the normal mode. The kernel executes in the system mode and thus has access to all segments, machine instructions, and hardware facilities. The

supervisor executes in the normal mode, and thus only has access to a subset of the instruction set and segments. The supervisor does not have access to those instructions which manipulate the system hardware, such as special I/O and execution mode control instructions.

4. Segmentation

Segmentation is the key element of a secure system. A segment is a logical grouping of information such as a procedure, array, or data area [8]. The address space of a process consists of those segments that may be addressed by that process. Segmentation is the management of those segments within the address space. In order to address a specific location within a segment two dimensions are required, an identification of the segment (e.g., segment number) and an offset from the base of the segment.

Each segment may have several logical attributes associated with it. These attributes can include segment size, classification, and access permitted (read, write, execute). The physical attributes of a segment include the current base address, and whether or not the segment is "in core". The segment's attributes and its physical location in memory are contained in a segment descriptor. The segment descriptors for a process are often contained in a descriptor list (viz., an MMU image for the SASS) to facilitate the memory management of its address space.

Segmentation permits multiple processes to share a single segment and to avoid the requirement of maintaining duplicate copies in memory. This eliminates the possibility of having conflicting data when multiple copies of the same segment are maintained. Segmentation also enables the enforcement of controlled access to a particular segment, since each process can have different access (read/write) to stored segments. This capability of enforcing controlled access is crucial to security.

Segmentation provides a mechanism for the virtualization of memory (although not provided in the SASS). If a user requests access to a segment to which he has access rights, and that segment is not in main memory, a memory fault will occur which will cause that segment to be loaded into main memory (another segment may have to be moved to secondary storage to make room). Thus to the user, the size of main memory is virtualized into the size of the process' address space.

5. Information Security

As previously stated, there is an ever increasing demand for a computer system to provide for the secure storage of information. This security cannot be added to an existing operating system with a large degree of confidence that the resulting security system cannot be avoided or bypassed. In order to be demonstrably adequate, security

must be designed into the operating system, and must be part of the cornerstone upon which the operating system is built.

There are two basic aspects of information security, external security and internal security. External security prevents an infiltrator from getting to the object in which the desired information is stored. This can be of such form as a fence, a safe, a sentry, or a guard dog. If an infiltrator manages to penetrate these external security measures, he then has access to the desired information. Internal controls would consist of those security measures internal to the computer which impede and if effective, prevent a compromise of information. If the internal controls function properly, information is provided and exchanged only with the users who are explicitly authorized access to that information. Many information systems are required to store and access information of different security levels (e.g., secret files interspersed with confidential and unclassified). The internal security of such a "multilevel" system must permit users and information to exist simultaneously at different security levels, and also ensure that no unauthorized accesses (either intentional or unintentional) are permitted. The SASS was designed to provide a multilevel secure storage environment.

The data to be stored in a secure information system can be looked upon as a set of logical objects such as files or records. Associated with each of these objects is a set

of subjects which have access rights to that object. These access rights may include read access, write access, or a combination thereof. The non-discretionary security policy involves checking the object's access class (oac) with the subject's access class (sac) to ensure that they are compatible. The access permitted is defined in a lattice model of secure information flow [9] as follows:

sac = oac, read and write access permitted

sac > oac, read access permitted

sac < oac, no access permitted

The government security classification system provides an example of a non-discretionary security policy. A user with a security clearance of confidential is authorized read and write access to a confidential file (sac = oac), and he has read access (but not write) to an unclassified file (sac > oac). This restriction on write access is to prevent the inadvertant writing of confidential data into an unclassified file to which the subject may have simultaneous access (this property is often referred to as the *-property [10]). Finally, the confidential subject does not have access to any secret files (sac < oac).

The discretionary security policy involves checking the subject against an object's access control list (ACL). The subject only has access to an object if he is included in its ACL. This policy is analagous with the government's "need to know" policy, which precludes a subject with a

secret clearance from having access rights to all secret information within the system. He may access only that for which he has a "need to know". The discretionary security policy thus allows the users of the system to specify who has access to their files. It is noted that the discretionary security policy is a refinement of the security policy, and never permits a violation of the non-discretionary security policy in effect.

The SASS was designed with the internal non-discretionary security to be provided by the security kernel. Discretionary security is provided by the supervisor file system. The security kernel is based upon a mathematical model which has been proven correct. This mathematical model implements the system's security policies.

The security kernel design has three prerequisites in order to provide a secure environment: 1) the kernel must be isolated to ensure that it cannot be modified either intentionally or inadvertantly. This is to ensure that the behavior of the kernel cannot be modified. 2) Each and every attempt to access data within the system must invoke the kernel. 3) The kernel's correctness must be verifiable. This implies that the mathematical model must be proved and demonstrated as secure, and that the kernel implements this model.

C. THESIS STRUCTURE

This thesis presents the detailed design of a memory management process for the SASS. The top down design technique was utilized, with levels of abstraction used to reduce the design complexity. The high level language utilized was PLZ/SYS, which was designed to be compatible with the Z8001 microprocessor. PLZ/SYS is a block structured language similar to PASCAL. The compiler which compiles from PLZ/SYS to the Z8001 instruction code is still in the developmental stage at ZILOG, INC. The PLZ/SYS code had to therefore be "hand compiled" (viz.,translated to the PLZ/ASM assembly language) in order to run, test, and debug the code. Some of the procedures in the lower levels of design (those which use privileged instructions to directly manipulate the system hardware) must be directly coded using the assembly code PLZ/ASM. These procedures were declared external to the Memory_Manager_PLZ/SYS_Module and are coded in the Memory_Manager_PLZ/ASM_Module.

Chapter II of the thesis presents an overview of the SASS at its current stage of development. The design of the memory management process, and the concurrent implementation of the distributed kernel processor management by Reitz [4] refined the original design of Parks and Coleman. Future work in the SASS will most likely require some refinement of the present design.

Chapter III presents the detailed design of the memory manager module. This chapter emphasizes why certain design features were chosen, and how they were implemented in this design.

The final chapter presents the status of research to date, and attempts to identify what follow-on work is required. The PLZ/SYS code module and the PLZ/ASM code module are presented as appendices.

II. SECURE ARCHIVAL STORAGE SYSTEM DESIGN

This chapter presents an overview of the SASS in its current state of development. It is a summation of the original design efforts, and reflects refinements of those original designs. This overview is necessary in order to fully understand the interrelationship between the memory manager and the overall system design. It also provides a current base for further SASS development.

A. BASIC OVERVIEW

The purpose of the SASS is to provide a secure archival file storage medium for a variable number of host computers. The key design goals of the SASS were multi-level internal computer security and controlled sharing of data among authorized users.

Figure 1 provides an example of how the SASS could be used. In this example, there are four host computers which reside in four separate rooms (consider each of these computers to be microcomputers, although any computer could be utilized). Each of the four hosts are used to create and manipulate files of fixed predetermined security classification. For example, all files created by host #2 are classified secret. Host #2 cannot create top secret,

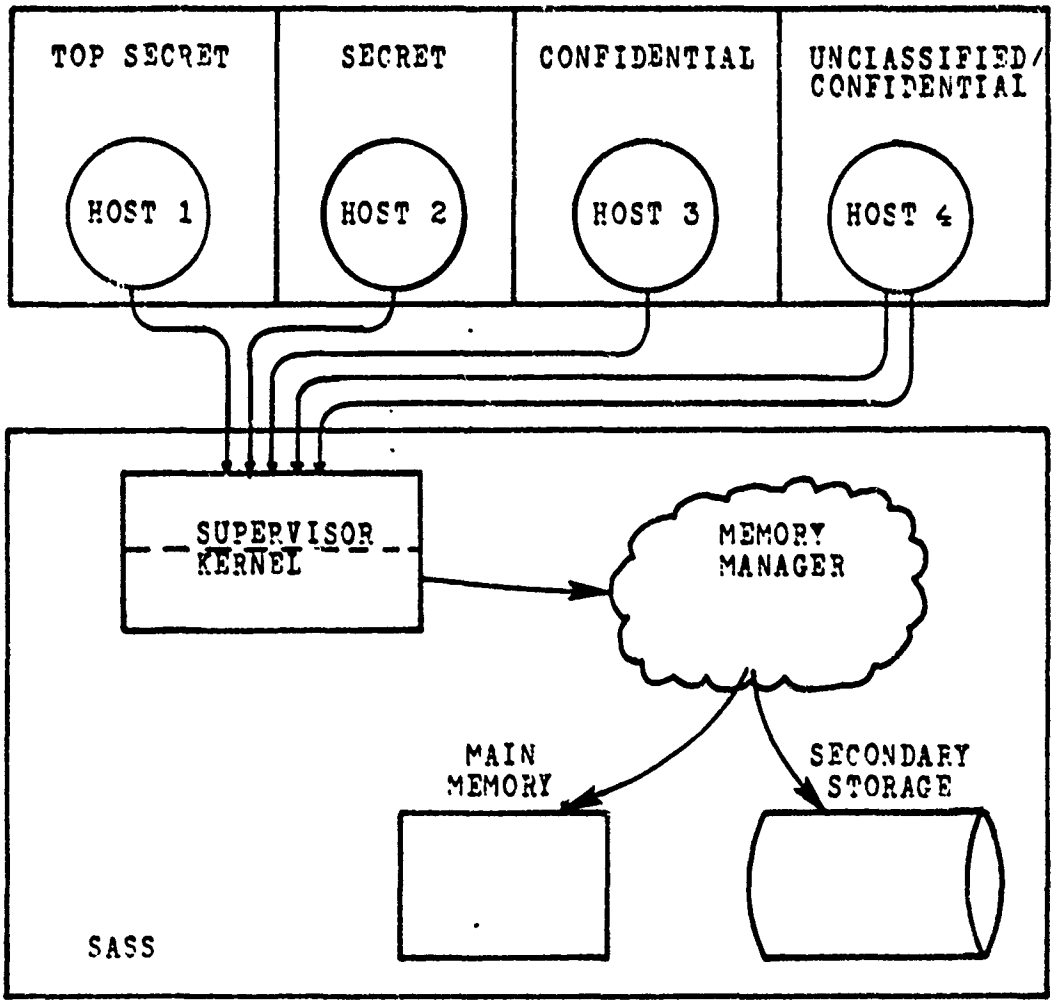


Figure 1. SASS System

confidential, or unclassified files (nor can he access top secret in this example). Access to each of these rooms is physically controlled to ensure that only personnel with the proper security clearance are authorized access. None of the host systems have a permanent local file storage device, and all are hard-wired to an I/O port of the SASS.

Each host controls the access to its I/O ports (host #4 illustrates the multi-level host connection currently required by the SASS). The physical protection of the hard-wire is assumed to be adequate to minimize the possibility of such malicious activities as wire tapping or emanations monitoring. Once a user of the host system completes his work, he can permanently store his file on the SASS, which is contained in the fifth room of figure 1 (view the SASS as an Z8001 microcomputer with access to secondary storage devices). To gain access to a file, the user or O/S of the host system must request the SASS to provide him with that file. This implies that if a malicious user gains access of the confidential host system, he still cannot access files of a higher classification.

The SASS must be capable of performing three basic functions in this environment. These functions are: 1) store a file for a host system, 2) retrieve a file for a host system, and 3) ensure that the the files are made available only to authorized users. The required capability of file storage and retrieval implies that processes must exist for

each host system to perform file management and data transfer on behalf of that host. To ensure the security of the stored information, the SASS must ensure that the user of a specific host system may only address the files to which he has access. The SASS achieves the desired environment through a distributed operating system design which consists of two primary modules, the supervisor and the security kernel (the security kernel actually consists of distributed and non-distributed portions). Each host system, which is hardwired to the SASS, communicates with its own I/O process and file manager process in the SASS itself.

The supervisor is responsible for the SASS-host system interface. It constructs and manages a hierarchical file system for its host, based upon the files which the host has submitted, and controls the actual I/O (both data and commands) between the SASS and the host system. The supervisor is built upon the security kernel and performs the host's requests (file storage, file retrieval, I/O) by calls to the security kernel. These calls must be validated (by a gate keeper module in the SASS design) before the security kernel function is invoked.

The SASS security kernel consists of a distributed and non-distributed kernel. The distributed kernel is distributed to (viz., is in the address space of) every process, and is responsible for the multiplexing of the

several processes onto the actual hardware processor(s), enforcing the non-discretionary security policy, and providing the synchronization primitives for inter-process communication. The non-distributed kernel consists of the memory manager process which is responsible for the secure management of both main memory and secondary storage. Each hardware processor must have its own memory manager (ergo, non-distributed kernel) in the SASS design.

An abstract system overview of the SASS is presented in figure 2. Four levels of abstraction were utilized to simplify the design and understandability of the system.

Level 0 consists of the system hardware which includes the Z8001 microprocessor, the local and global memories, and secondary storage. The SASS is designed to operate in a multi-microprocessor environment, therefore each CPU is assigned its own local memory (to which it alone has access) in which it can store process local segments. The system contains a global memory, which every CPU may access. Segments to which a user process has write access must be stored in global memory if more than one process has simultaneous access to that segment. This is to ensure that all processes access the current copy of that shared writable segment. The basic storage policy is to store every segment within local memory if at all possible. This is to keep bus contention between processors, which access global memory, to a minimum.

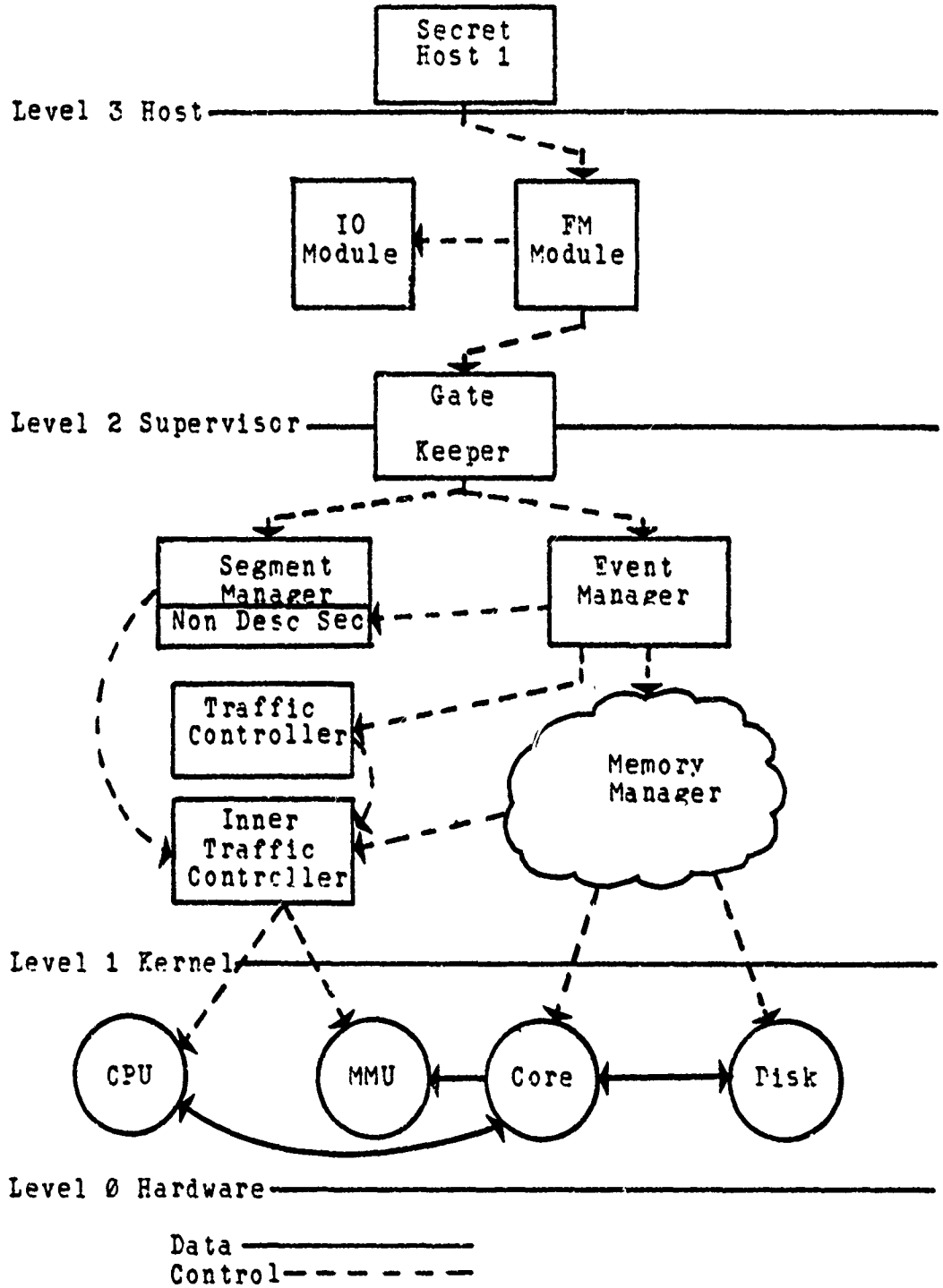


Figure 2. SASS Abstract System Overview

Level 1 consists of the distributed and non-distributed kernel. The kernel is placed in (executes in) the most privileged domain (system mode) of the Z8001 to ensure that it is protected from any manipulation (either malicious or inadvertant). The kernel controls all access to the system hardware by maintaining all privileged machine instructions within its domain. Only the kernel may access these instructions. The distributed kernel is responsible for creating a virtual processor environment and enforcing the non-discretionary security policy. It multiplexes processes onto virtual processors and then multiplexes these virtual processor(s) onto the actual hardware processors. The non-distributed kernel consists of the memory manager and is responsible for the secure management of both main memory and secondary storage.

Level 2 consists of the supervisor, which resides in the less privileged domain (normal mode) of the Z8001 microprocessor. It has access to all the machine instructions with the exception of those which manipulate the system hardware. The supervisor must request the kernel to move segments into and out of memory and secondary storage via the gate keeper (a software assisted ring-crossing mechanism). The supervisor consists of two surrogate processes for each host, the I/O (input/output) process and the FM (file management) process. By utilizing the I/O and FM processes the supervisor is able to provide

and manage a virtual file hierarchy for each host system. Each host system has I/O and FM processes created and assigned at system generation. They are not dynamically created or deleted. The supervisor ensures that each segment's discretionary security is enforced.

Level 3 consists of the host computer systems. These systems are hardwired to the I/O ports of the Z8000. The hosts communicate with the SASS via system protocols over a communication link. Any computer system could serve as a host, with each host supporting multiple users.

3. SUPERVISOR

Each host system is assigned the dedicated services of a pair of supervisor processes at system generation. These processes are the I/O and FM processes. The FM process and the I/O process communicate with each other via a shared segment entitled the "mailbox". This communication is synchronized via the kernel synchronization primitives which act upon eventcounts and sequencers [12]. A virtual file system is created and maintained for each host by its FM and I/O processes.

1. File Management Process

The FM process is responsible for the management of the host's virtual file system within the SASS. The FM

process interprets all the host commands and acts upon them in conjunction with the I/O process.

The user of the host system views his stored data (within the SASS) as a hierarchy of files. Figure 3 provides an example of such a hierarchical file structure. To specify a particular file, a pathname is required. The pathname is simply a concatenation of the file names (given to each file by the user at its creation) starting at the "root" directory and proceding sequentially to the desired file. The user is required to submit a pathname with each command sent to the SASS. The five basic actions to be performed upon files at this level are: 1) to create a file (data or directory), 2) to delete a file, 3) to read a file (data or directory), 4) to initiate or modify file attributes (size, classification, access permitted), and 5) to store (write) a file.

The FM process is required to convert the pathname provided by the user, into one or more segment numbers. This is necessary because the notion of a file is not known within the kernel. All files are composed of segments, and must be referenced as segments within the kernel for manipulation and management. The FM process must also provide appropriate command handlers to ensure that the user's requested action is properly carried out.

The SASS permits a host to read or write the files of another host, at the same security level, if

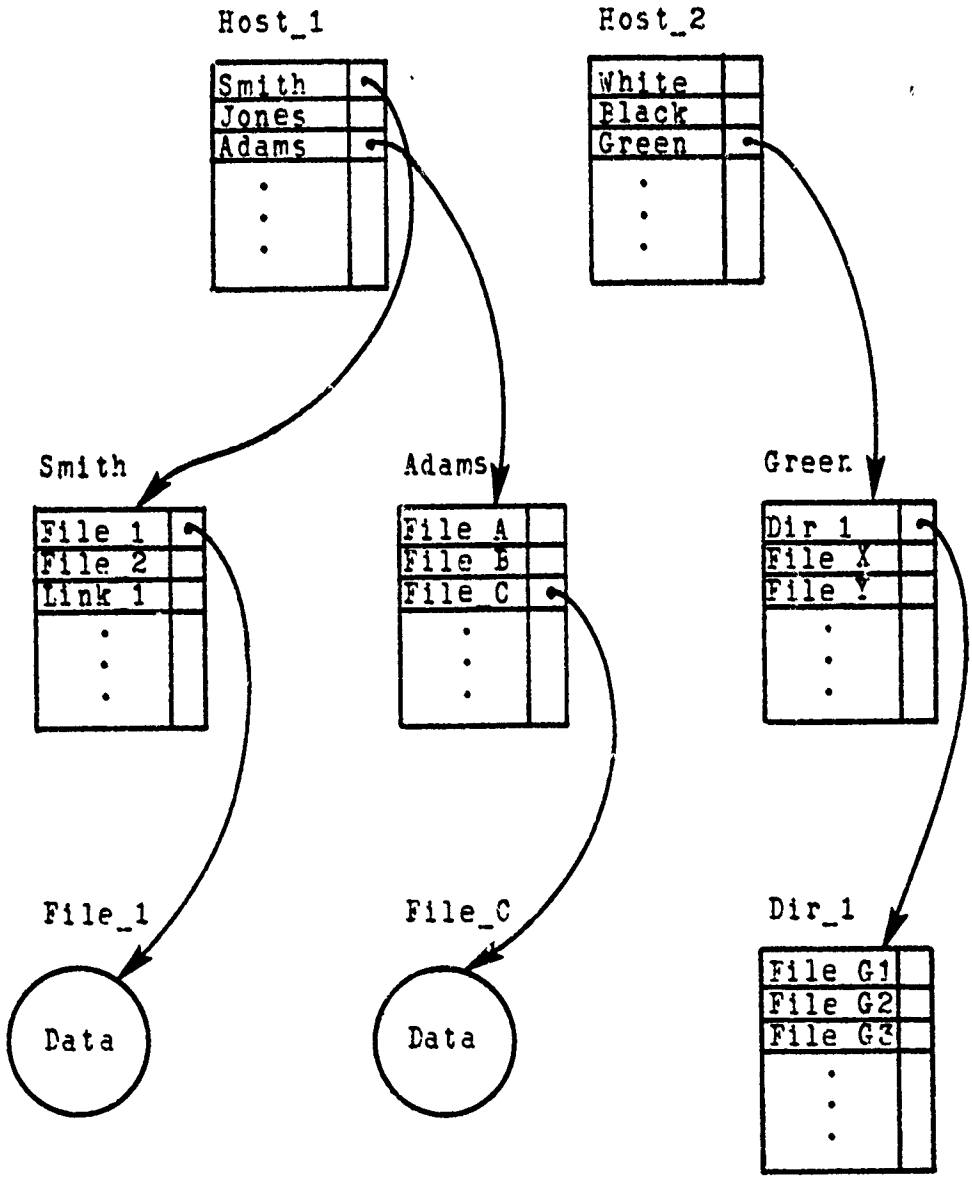


Figure 3. Virtual File Hierarchy

discretionary access is permitted. Files of a lower classification may be read only (if discretionary access is permitted). This file sharing is achieved by creating a link between the two file hierarchies. This link is entered into a directory file of the host, and is constructed in the same manner as a pathname (viz., it is a concatenation of filenames). The kernel enforces a read only access to the lower classified files, which prevents the possibility of writing data (through a link) of a higher classification into a file of lower classification.

The database utilized by the FM process to manage the host's files is the FM Known Segment Table (FM_KST). The FM_KST is a list of those segments which are known to (viz., within the address space of) the FM process. Figure 4 provides an example of the FM_KST structure.

Whenever a user of a host system requests access to a specific file, the FM_KST is searched to determine if that pathname (segment) is already known. If it is known, the request is passed to the kernel, via the gate_keeper, with the appropriate [illegible]cut number, for the desired action. If the pathname is not known, the segment number of the desired file's directory (parent) file and an entry number are sent to the kernel with the request to make that segment known. If the request is authorized by the kernel, a segment number and access mode authorized are returned. The returned segment number and mode are then entered into the FM_KST

Path Name	Seg_#	Access Mode	Use
Host_1>Adams>File_C	50	R	N
Host_2>Green>Dir_1	44	W	Y
Host 1>Smith>File_1	22	W	N
Host_1>Smith>Link_1	44	R	Y
. . . . .			

Figure 4. File Manager Known Segment Table.

with that segment's pathname. Once the segment is known, the desired user action can be carried out.

The user requests to create or delete files are simply passed to the appropriate kernel procedure, via the gate keeper, by the FM process (after a discretionary security check). No entries are added or deleted from the FM_KST during create or delete requests (they invoke kernel primitives which add or delete entries from a kernel data base).

Should the FM process request that a segment be swapped into memory and memory is full, an error code will be returned to the FM process from the kernel (it is noted that this is a per process memory allocation, thus the memory state cannot be affected by its use by other processes). The FM process will then select a segment to be removed from core to make room for the desired segment. The current design calls for the invocation of a least recently used algorithm (LRU) which makes use of the FM_KST "used" field to determine the least recently used segment for swap out.

Discretionary security is enforced in the discretionay security module of the FM process. An access control list (ACL) is maintained for each file within the file hierarchy. The ACL is simply a list of authorized users (a refinement of non-discretionary security) which is checked for each access to that file. The discretionary

security module also performs the housekeeping functions for the file's ACL. These functions include the addition of a ACL entry, the deletion of an ACL entry, and the initialization of an ACL for a new file.

It is noted that the original design of the FM process contained a memory manager procedure. This was necessary because the original SASS design called for the partitioning of memory such that each supervisor maintained his own core. The FM memory manager managed this virtual core by calls to the kernel via the gate keeper (swap_in, swap_out). The current design of the non-distributed kernel includes memory allocation and thus has removed the need for the supervisor to manage its own virtual core. Because of this, a FM memory manager is not required.

2. Input/Output Process

The I/O process is responsible for all the input and output between the supervisor and the host computer system. The I/O process receives its commands from the FM process via the shared mailbox segment.

Data is transfered between the host systems and the SASS in fixed size "packets". There are three basic types of packets, a synchronization packet, a command packet, and a data packet. Protocols exist for the reliable transmission and receipt of the packets by both the SASS and the host systems. The current design calls for the use of

multi-packet protocols, which allows the sender to send several packets before he receives a receipt.

The original design of the I/O process contained a Memory Manager procedure for the same reasons as the FM process. This procedure is no longer required due to the design of the non-distributed kernel.

C. DISTRIBUTED KERNEL

The initial design of the security kernel as presented by Coleman [3], has been developed by Reitz [4] and the work presented here. The primary refinements have been the replacement of block/wakeup [3] by eventcounts, the inclusion of an event manager which contains the synchronization primitives, and the trarsfer of MMU management to the memory manager. Figure 5 provides an overview of the security kernel design.

1. Gate Keeper

The gate keeper is a software ring crossing mechanism which is utilized to ensure that the security kernel is isolated and tamperproof. The major issues of the gatekeeper design are: 1) to provide a mode switching mechanism for switching from normal (supervisor) mode to system (kernel) mode , 2) to mask hardware preempt interrupts in the kernel, and 3) to check for "virtual"

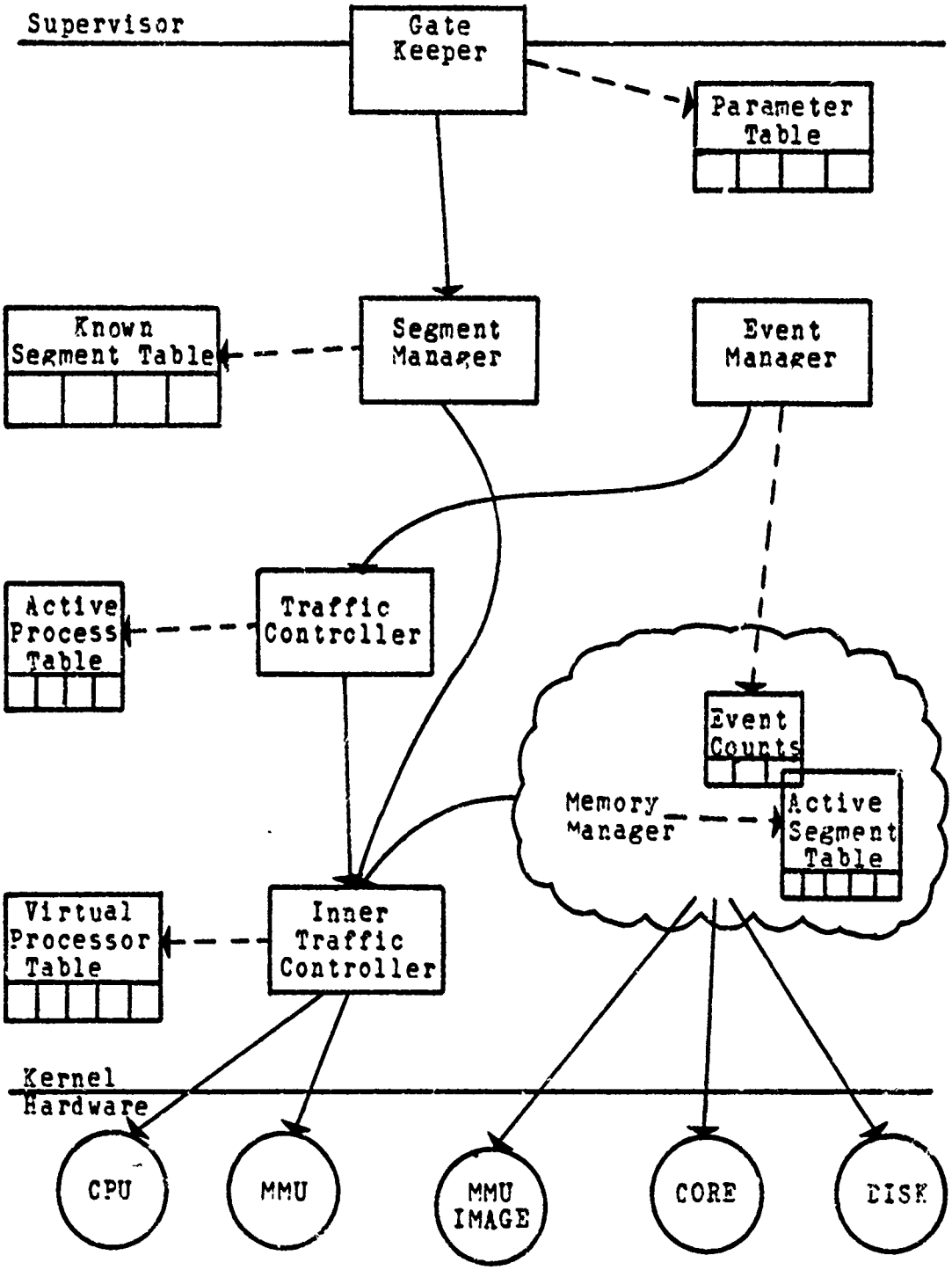


Figure 5. Security Kernel Design.

software preempt interrupts when leaving the kernel. The gate keeper provides the sole entry point into the kernel domain, validates the request and its arguments, and transfers the request to the appropriate kernel procedure. If the gate keeper encounters an error, it returns an appropriate error code without irvoking the kernel.

The gate keeper uses a parameter table to validate the user's request (call by value only). This table contains the number of parameters required by each kernel function (create_segment,delete_segment, etc.), the type of each parameter, and the type of each return parameter. If an error is discovered during the validation process, it sets the return message to an error code. If the request is valid, the gate keeper calls the appropriate kernel module.

The gate keeper is a trap handler. The supervisor puts an argument list and space for a return message in a segment (or processor registers) within the supervisor's domain. When the gate keeper is invoked, it must first save the supervisor processor registers and then retrieve the argument list (via an argument list pointer register). The arguments are validated and if correct, passed to the appropriate kernel module.

When the kernel completes action taken upon the user request, it returns to the gate keeper. The gate keeper then copies a return message into the return argument (that is returned to the supervisor's domain), restores the

supervisor's environment, unmasks the interrupts, and makes a trap return back to the supervisor (viz., changes the mode back to normal).

2. Segment Manager

The segment manager is responsible for the management of the segmented virtual memory. There are six functions which the segment manager is called upon to perform. These functions are: 1) to create a segment, 2) to delete a segment, 3) to make a segment known, 4) to make a segment unknown (terminate), 5) to swap a segment into core, and 6) to swap a segment out of core.

The segment manager uses the Known Segment Table (KST) as its data base to manage segments. The KST is a process local kernel data base which contains entries for all the segments which the process has made known. Figure 6 provides an example of the KST structure. The KST size is fixed at system generation. It is indexed by segment numbers which are assigned by the segment manager. When a segment is made known, a "handle" (the concatenation of the Global Active Segment Table (G_AST) index and the segment's unique identification) is returned to the segment manager by the memory manager. The handle is a system wide unique identification that is assigned to each active segment (viz., active in the G_AST). The KST provides the mapping mechanism for converting the segment number into the

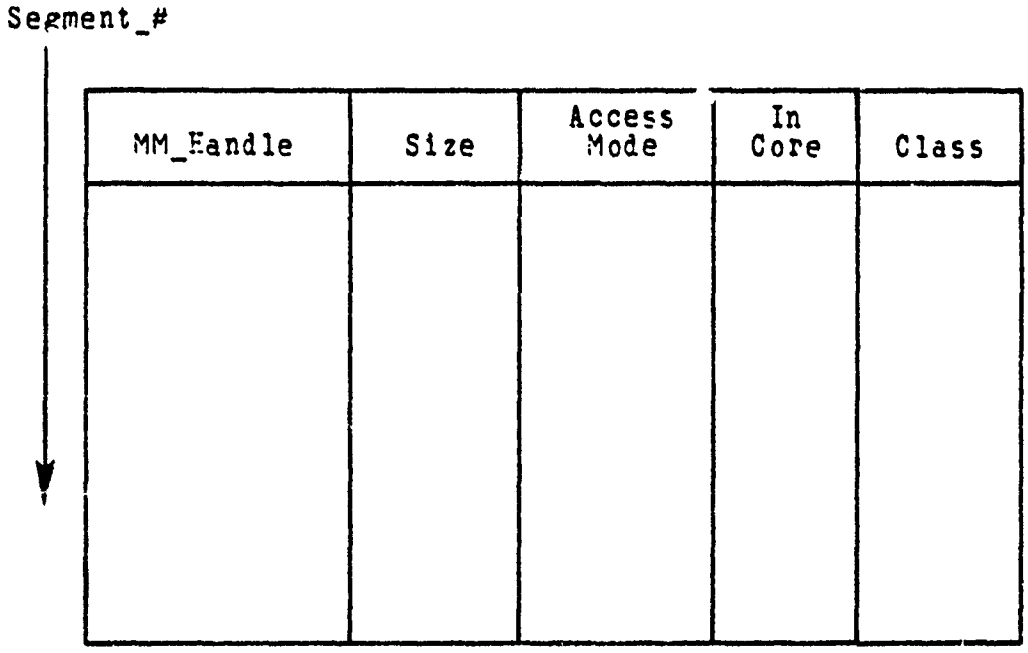


Figure 6. Known Segment Table.

segment's unique handle. The use of the unique handle by the memory manage. is what permits the controlled sharing of segments by concurrent processes. Any process which requests to make a specific segment active will always be returned that segment's unique handle. Thus any one segment may exist within the address space of several processes (with a different segment number in each process) while residing in one location in memory.

The SIZE field of the KST represents that segment's size. Segments exist in multiples of 256 bytes due to Z-8000 MMU hardware constraints. An upper bound upon the segment size is fixed at system generation by the design parameter max_segment_size. This is limited to 65K bytes by hardware. The ACCESS_MODE field states the access authorized to the segment (read, write) by this process. The IN_CORE field is set when a process successfully requests the segment to be swapped into core. The CLASS field is used to give the access class (e.g., secret, confidential) of the segment.

The usual sequence of invoking the segment manager functions (by the supervisor) would be as follows: 1) Create_Segment (this will invoke the memory manager to assign a unique identification to the created segment), 2) Make_Known, which will place the segment into the KST, and 3) Swap_In, which will move the segment from secondary storage to main memory. To remove a segment from main memory

to secondary storage, the order would be 1) Swap_Out, 2) Make_Unknown, and 3) Delete_Segment.

3. Event Manager

The event manager provides the kernel synchronization primitives that are used for the synchronization of concurrent processes in the supervisor of the present SASS design. The synchronization mechanism used is that of eventcounts and sequencers, first proposed by Reed and Kanodia [10]. The use of eventcounts and sequencers allows the ordering of events to be controlled directly by the processes involved, rather than to depend upon mutual exclusion mechanisms such as semaphores. The actual eventcounts are maintained in the memory manager module as they are a system wide entity and are not process local.

Reed and Kanodia define an eventcount as an object that keeps a count of the number of events in a particular class that have occurred so far in the execution of the system. The event observed can be anything from the input of data to the system, to writing a particular segment. The eventcount can be viewed as an integer value, which is incremented with each occurrence of the observed event. The primitive ADVANCE(X) is used to signal the occurrence of a particular event, and causes the eventcount X, associated with that event, to be incremented. The primitive READ(X) will return the value of the eventcount X. The primitive

AWAIT(X,n) will suspend the calling process until the value of eventcount X is greater than or equal to the integer value n.

A sequencer can be defined as an abstract object that can be utilized to totally order the events of a particular class. The basic purpose of the sequencer is to provide a means to determine an ordering of a set of occurences of a particular event. Like the eventcount, the sequencer can be viewed as an integer value which is incremented each time the primitive TICKET(S) is called. The TICKET primitive is based upon the ticket machines often used in barbershops and ice cream stores. When a customer enters, he takes a ticket, from which the order of who arrived first and whom will be served next can be determined.

The use of eventcounts and sequencers by the SASS supervisor can be illustrated as follows. Suppose that segment A is currently being updated by process one. Eventcount A currently has the value of 9 (the eventcount associated with the reading of segment A). Process two desires to read segment A, so he obtains a ticket by utilizing the TICKET primitive associated with segment A. The value returned by TICKET is 10. Process two now calls upon the primitive, AWAIT(A,10), which will suspend process two until eventcount A is valued at 10. When process one completes his update, he will execute ADVANCE(A), which will

increment eventcount A to the value of 10. This will allow the AWAIT(A,10) to return to process two, which will then be allowed to read segment A.

4. Traffic Controller

The traffic controller performs the function of scheduling processes to run on virtual processors. The traffic controller could be designed to schedule processes to run directly on the hardware processors, but in this design, Reed's [11] notion of a two level traffic controller was utilized. Thus the processes are first multiplexed onto virtual processors by the traffic controller. The virtual processors are then multiplexed onto the actual hardware processors by the inner traffic controller.

A virtual processor is an abstract data structure which preserves all the attributes of a process in execution on a processor (i.e., an execution point and an address space). Multiple virtual processors may exist for a single physical processor. The Active Process Table (APT) is the data base utilized by the traffic controller to control and manage the multiplexing of processes onto virtual processors. Figure 7 provides an example of the the APT.

The APT is a fixed sized table which contains an entry for each process of the SASS (the processes are created at system generation). Because of the design decision not to create or destroy processes after system

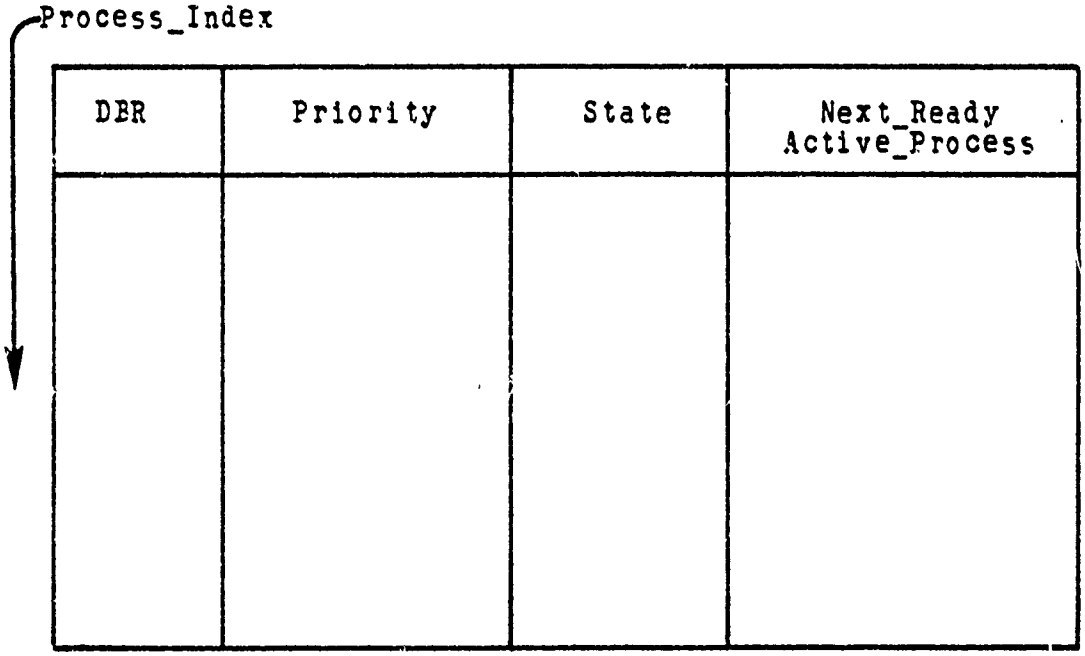


Figure 7. Active Process Table.

generation, the initial entries into the APT will be active for the life of the system. The index into the APT is the PROCESS_ID.

The traffic controller uses the PRIORITY field of the APT to determine which process to schedule for execution on each virtual processor. The STATE field contains that process' current state (running, blocked, or ready). The DBR (descriptor base register) field of the APT provides the address of the MMU image for that process. The Next_Ready_AP field is a pointer which contains the index of the next process which is in the ready state.

The design simplification choice of always having a process running on the virtual processors, introduced the notion of an idle process for each virtual processor. The idle process is loaded onto a virtual processor and placed into the running state whenever the number of available virtual processors exceeds the number of ready or running processes (excluding the idle process). The idle process is of the lowest priority, and will only run if no other process can be loaded. It is incapable of blocking itself, and thus must always be in either the running or ready state.

When a virtual processor becomes available, the traffic controller will be invoked to schedule the highest priority ready process which may run on that particular virtual processor. If no process is ready, the Idle process

is scheduled. The Idle process provides a means to guarantee that a ready process will always be found, and that the Traffic Controller cannot be exited without scheduling a process.

5. Inner Traffic Controller

The purpose of the inner traffic controller is to provide the multiplexing of the virtual processors onto the actual system processor(s), and to provide the kernel primitives for inter-process communication within the kernel (Signal and Wait). In the SASS design, each physical processor has a fixed set of virtual CPU's that it multiplexes. The primary data base utilized by the inner traffic controller is the Virtual Processor Table (VPT). Figure 8 provides an example of the VPT.

The VPT is indexed by the Virtual_Processor_ID. The DBR, PRI, and the STATE fields are used in the same manner as those fields in the APT. The Idle_Flag simply indicates that the idle process is loaded on that virtual processor. The Preempt flag indicates that a virtual preempt interrupt has been directed to that virtual processor. The Phys_Processor is a fixed field that indicates which hardware processor that virtual processor is scheduled to run on. The Next_Ready_VP is a pointer to the index of the next ready virtual processor in the VPT for this CPU.

In his original design, Coleman [3] tasked the inner

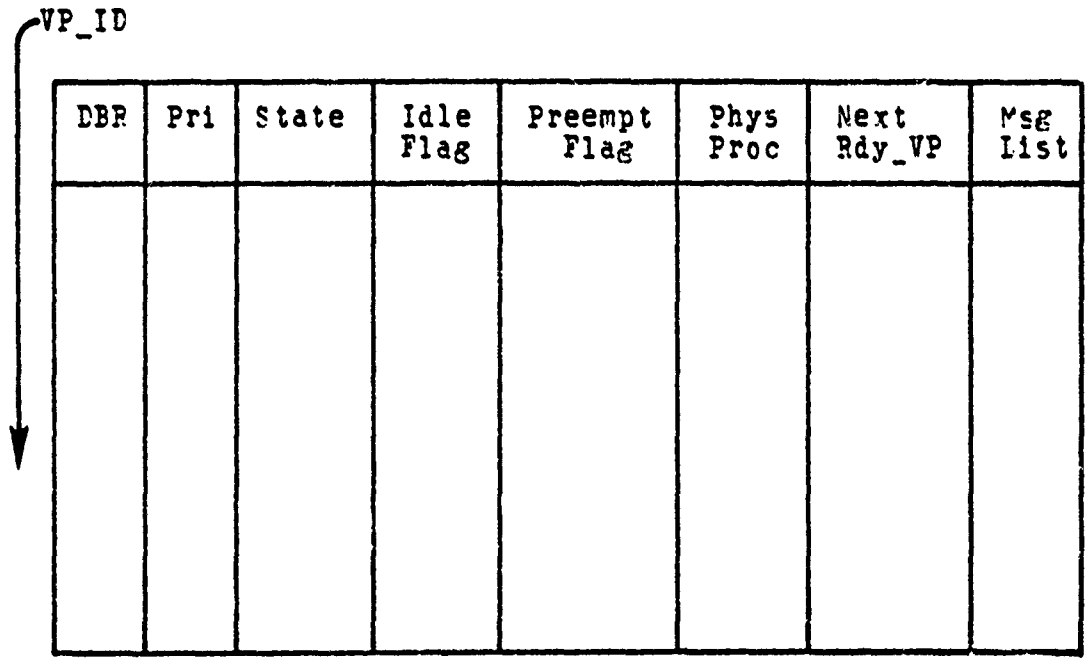


Figure 8. Virtual Processor Table.

traffic controller with the management of the hardware Memory Management Units (which contain the process' address space and its attributes) and the MMU software images. In the present design, this function has been assigned to the memory manager. When the inner traffic controller unloads a processor, it simply writes the MMU into the MMU image in order to save the segment usage information. To load a process, it writes the MMU image into the MMU. The memory manager insures that the MMU image is kept current by updating the images whenever a segment is swapped in or swapped out of memory.

The kernel synchronization primitives of SIGNAL and WAIT are maintained within the inner traffic controller. These primitives are used by virtual processors within the kernel domain to synchronize with other virtual processors within the kernel domain.

D. NON-DISTRIBUTED KERNEL

The SASS non-distributed kernel is composed solely of the memory manager process. Each physical processor has associated with it, its own dedicated memory manager process. The purpose of the process is the proper and secure management of the main memory (both local and global), and secondary storage. The actual transfer of segments from main memory to secondary storage and vice-versa, is controlled by

the memory manager process. The primary data base utilized by the process is the Active Segment Table. Chapter 3 provides a detailed description of the process' functions and data bases.

III. MEMORY MANAGER PROCESS DETAILED DESIGN

A. INTRODUCTION

The memory manager is responsible for the management of both main memory (local and global) and secondary storage. It is a non-distributed portion of the kernel with one memory manager process existing per physical processor. The memory manager is tasked (via signal and wait) to perform memory management functions on behalf of other processes in the system. The major tasks of the memory manager are : 1) the allocation and deallocation of secondary storage, 2) the allocation and deallocation of global and local memory, 3) segment transfer from local to global memory (and vice versa), and 4) segment transfer from secondary storage to main memory (and vice versa). There are ten service calls (via signal) which task the memory manager Process to perform these functions. The ten service calls are:

CREATE_ENTRY
DELETE_ENTRY
ACTIVATE
DEACTIVATE
SWAP_IN
SWAP_OUT
DEACTIVATE_ALL
MOVE_TO_GLOBAL
MOVE_TO_LOCAL
UPDATE

Upon completion of the service request, the memory manager returns The results of the operation to the waiting process

(via signal). It then blocks itself until it is tasked to perform another service. The hardware configuration managed by the memory manager process is depicted in figure 9. The shared data bases used by all memory manager processes are the Global Active Segment Table (G_AST), the Alias Table, the Disk Bit Map, and the Global Memory Bit Map. The processor local data bases used by each process are the Local Active Segment Table (L_AST). the Memory Management Unit Images and the Local Memory Bit Map.

F. DESIGN PARAMETERS AND DECISIONS

Several factors were identified during the design of the memory manager process that refined the initial kernel design of Coleman[3]. The two areas that were modified, were the management of the MMU images and the management of core memory. Both of these functions were managed outside of the memory manager in the initial design. The inclusion of these functions in the memory manager process significantly improved the logical structure of the overall system design. Additional design parameters were established to facilitate the initial implementation. These design parameters need to be addressed before the detailed design of the memory manager process is presented.

It was decided to make the block/page size of both main memory and secondary storage equal in size. This was to

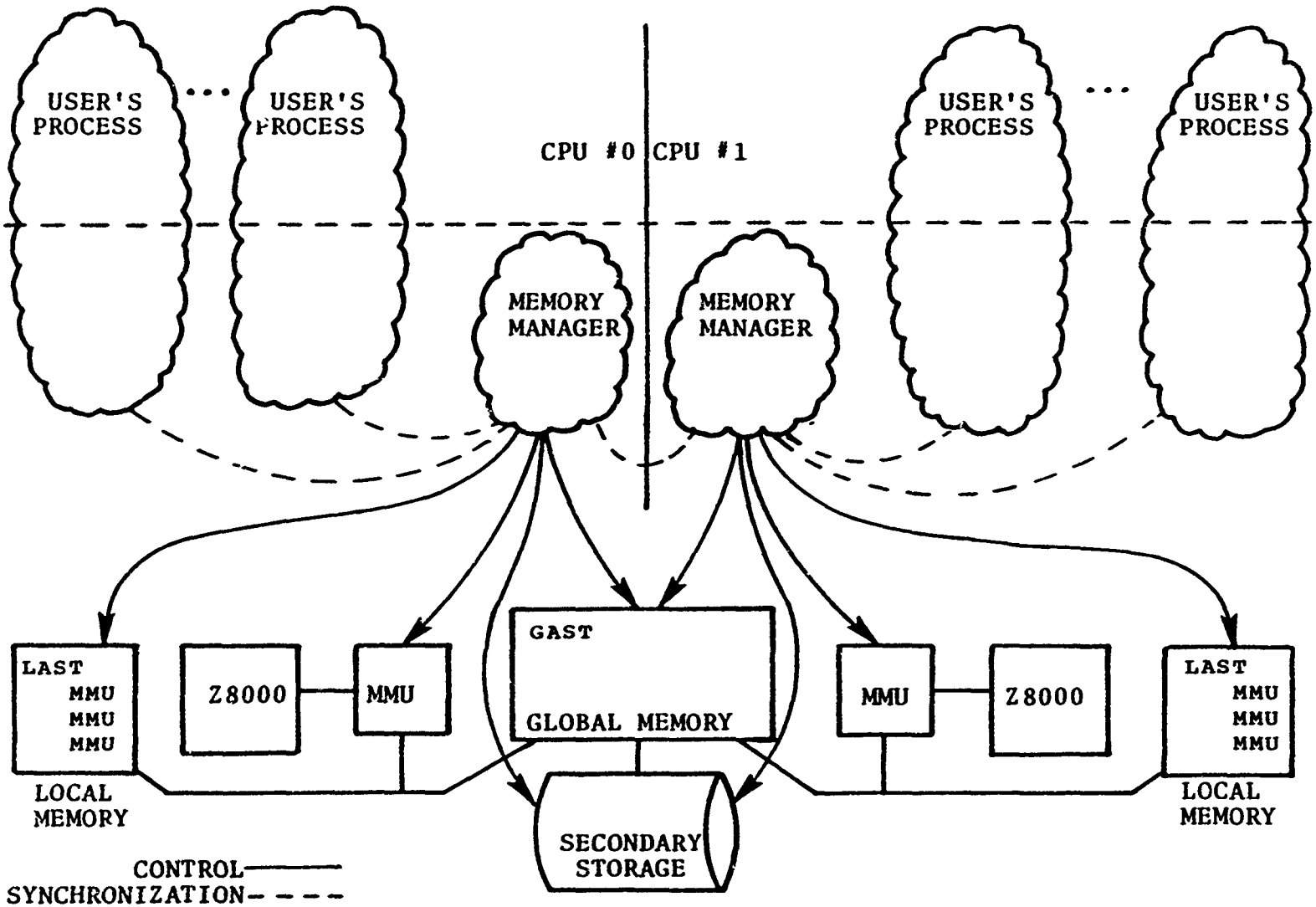


Figure 9. SASS H/W System Overview.

simplify the mapping algorithm from secondary storage to main memory (and vice versa). In the initial design the block/page size was set to 512 bytes.

The size of the page table for a segment was set at one page (non-paged page table). This was to simplify implementation, and had a direct bearing on the maximum segment size supported in the memory manager. For example, a page size of 256 bytes will address a maximum segment size of 32,768 bytes, while a page size of 512 bytes will address a segment size of 131,072 bytes.

The size of the alias table was set to one page (non-paged alias table). The number of entries that the alias table will support is limited by the size of the page table (viz., a page size of 512 bytes will support up to 46 entries in the Alias Table).

In the original design, the main memory allocation was external to the memory manager. This was due to the partitioned memory management scheme outlined by Parks[2] and Coleman[3]. In the current design, all address assignment and segment transfer are managed by the memory manager. This design choice enhanced the generality of the design, and provided support for any memory management scheme (either in the memory manager or at a higher level of abstraction). However, the current design still has a maximum core constraint for each process.

Dynamic memory management is not implemented in this design. Each process is allocated a fixed size of physical core. Fowever, it is not a linear allocation of physical memory. The design supports the maximum sharing of segments in local and global memory. All segments that are not shared, or shared and do not violate the readers/writers problem will reside in local memory to eliminate the global bus contention. The need to compact the memory (because of fragmentation) should be minimal in this design due to the maximum sharing of segments. If contiguous memory is not available, the memory manager will compact main memory. After compaction, the memory can be allocated.

The design decision to represent memory as one contiguous block (not partitioned) was made to support a dynamic memory management scheme. Without dynamic memory management, the process' total physical memory can not exceed the systems main memory. The supervisor knows the size of the segments and the size of the process' virtual core, therefore it can manage the swap in and swap out to ensure that the process' virtual core has not been exceeded.

In the original design, the user's process inner-traffic controller maintained the software images of the memory management unit. This design required the memory manager to return the appropriate memory management data (viz.,segment location) to the kernel of the user's process. In the current design, the software images of the MMU are

maintained by the memory manager. A descriptor base pointer is provided for the inner-traffic controller to multiplex the process address spaces. The MMU image data base does not need to be locked (to prevent race conditions) due to the fact that process interrupts are masked in the kernel. Thus, if the memory manager (a kernel process) is running then no other process can access the MMU image.

The system initialization process has not been addressed to date. However, this design has made some assumptions about the initial state of the system. Since the memory manager handles the transfer of segments from secondary storage to main memory, it is likely to be one of the first processes created. The memory manager's core image will consist of its pure code and data sections. The minimal initialization of the memory manager's data bases are entries for the system root and the supervisor's segments in the G_AST and I_AST(s), and the initializaton of the MMU images with the kernel segments. The current design does not call for an entry in the G_AST or I_AST for the kernel segments. However, when system generation is designed this will have to be readdressed.

The original[3] memory manager data bases have been refined by this thesis to facilitate the memory management functions. The major refinemerts of the global and local active segment tables are outlined in the following section.

C. DATA BASES

1. Global Active Segment Table

The Global Active Segment Table (see figure 10) is a system wide, shared data base used by memory manager processes to manage all active segments. A lock/unlock mechanism is utilized to prevent any race conditions from occurring. The signalling process locks the G_AST before it signals the memory manager. This is done to prevent a deadly embrace from occurring between memory manager processes, and also to simplify synchronization between memory managers. The entire G_AST is locked in this design to simplify the implementation (vice locking each individual entry).

The G_AST size is fixed at compile time. The size of the G_AST is the product of the G_AST record size, the maximum number of processes and the number of authorized known segments per process. Although the G_AST is of fixed size, it is plausible to dynamically manage the entries as proposed by Richardson and O'Connell[1]. The current memory manager design could be extended to include this dynamic management.

The Unique_Id field is a unique segment identification number in the G_AST. This field is four bytes wide and will provide over four billion identification numbers. A design choice was made not to manage the

Index_#

Unique ID	Global Addr	* Processors L_ASTE_#		Flag Bits		G_ASTE_# Parent
		#0	#1	Written Bit	Writable Bit	

* Field indicates a two processor environment

# Active In Memory	No. Active Depend.	Size	Page Table Loc	Alias Table Loc	Seq-uencer	Inst-ance1	Inst-ance2

Figure 10. Global Active Segment Table.

reallocation of the unique_id's. Thus when a segment is deleted from the system, the unique_id is not reused.

The Global_Address field is used to indicate if a segment resides in global or local memory. If not null, it contains the global memory base address of a segment. A null entry indicates that the segment might be in local memory(s).

The Processors_L_ASTE_# field is used as a connected processors list. The field is an array structure, indexed by Processor_Id. It identifies which L_AST the segment is active in, and provides the index into each of these tables. The design choice of maintaining an entry in the L_AST for all locally active segments implies that if all entries in the Processors_L_ASTE_# field are null, the segment is not active and can be removed from the G_AST (viz., no processors are connected).

The Flag_Bits field consists of the written bit, and the writable bit. The written bit is set when a segment is swapped out of memory, and the MMU image indicates that it has been written into. The writable bit is set during segment loading to indicate that some process has write access to that segment.

If an active segment is a leaf, the G_ASTE_#_Parent field provides a back pointer to the G_AST index of its parent. This back pointer to the parent is important during the creation of a segment. If a request is received to

create a segment which has a leaf segment as its parent, then an alias table has to be created for that parent. Also, the alias table of the parent's parent needs to be updated to reflect the existence of the newly created alias table (see figure 11). The indirect pointer shown is the back pointer to the parent via the G_AST.

The No_Active_In_Memory field is a count of the number of processes that have the segment in global memory. It is used during swap out to determine if the segment can be removed from global memory.

The No_Active_Dependents field is a count of the number of active leaf segments that are dependent on this entry (viz., require that this segment remain in the G_AST). Each time a process activates or deactivates a dependent segment this field is incremented or decremented.

The Size field is the size of the segment in bytes. The Page_Table_location field is the disk location of the page table for a segment, and the Alias_Table_Location field is the disk location of the alias table for the segment. The Alias_Table field can be null to indicate that no alias table exists for the segment.

The last three fields are used in the management of eventcounts and sequencers [4]. The Sequencer field is used to issue a service number for a segment. The Instance_1 field and Instance_2 field are eventcounts (i.e., are used to indicate the next number of occurances of some event).

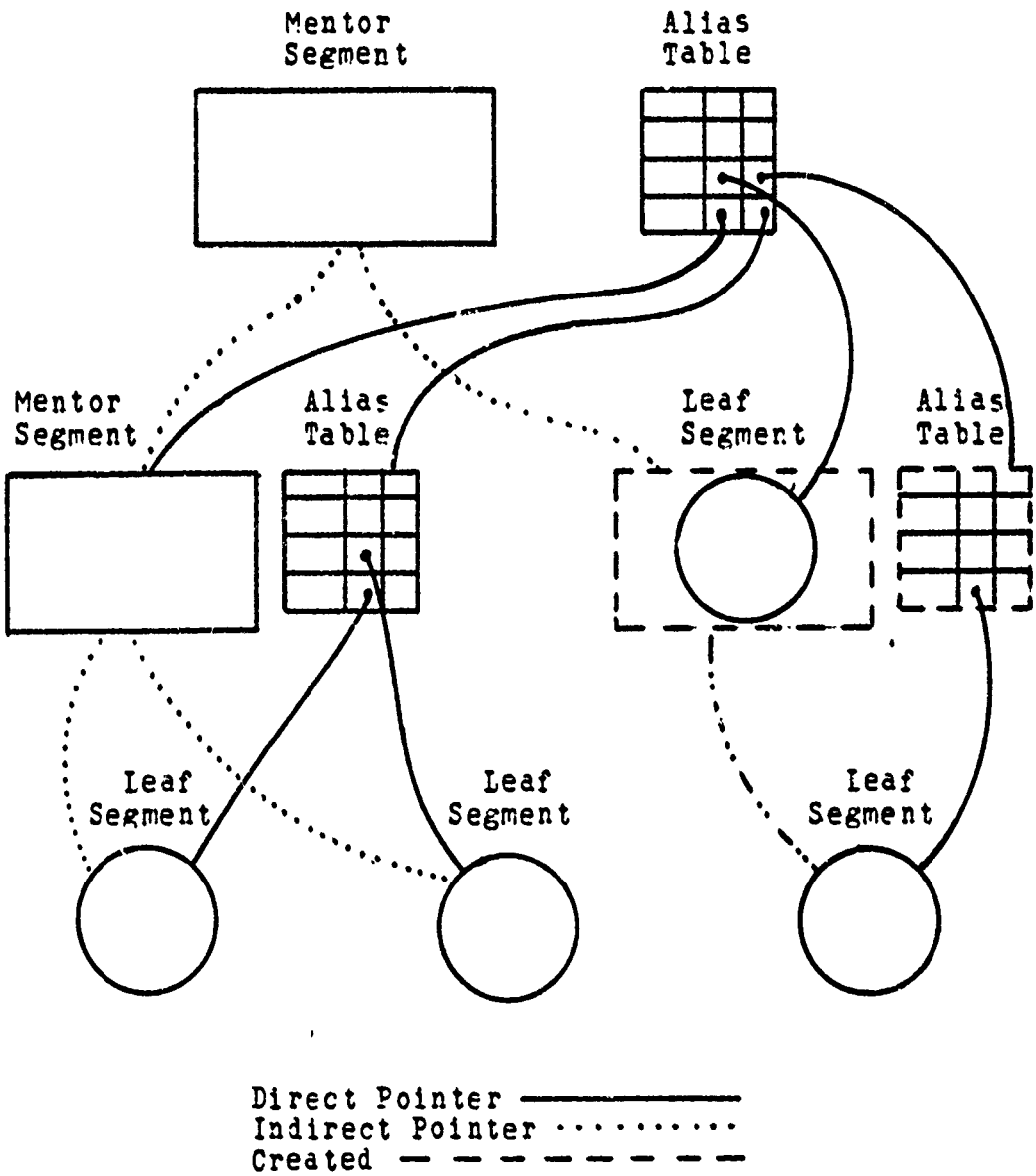


Figure 11. Alias Table Creation.

2. Local Active Segment Table

The Local Active Segment Table (see figure 12) is a processor local data base. The L_AST contains the characteristics (viz., segment number, access) of each locally active segment. An entry exists for each segment that is active in a process "loaded" on this CPU and in local memory. The first field of the L_AST contains the memory address of the segment. If the segment is not in memory, this field is used to indicate whether the L_AST entry is available or active. The Segment_No/Access field is a combination of segment number and authorized access. It is an array of records data structure that is indexed by DSR_*. The first record element (viz.,most significant bit) is used to indicate the access (read or read/write) Permitted to that segment. The second record element (viz., the next seven bits) is used to indicate the segment number. A null segment number indicates that the process does not have the segment active.

3. Alias Table

The alias table (see figure 13) is a memory manager data base which is associated with each non leaf segment in the kernel. An aliasing scheme is used to prevent passing systemwide information (unique_id.) out of the kernel. Segments can only be created through a mentor segment and

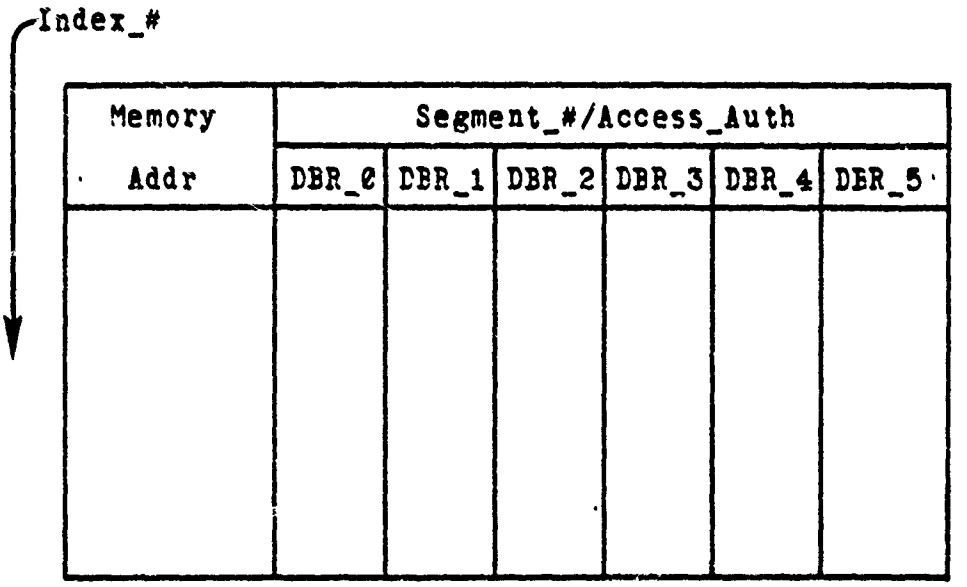


Figure 12. Local Active Segment Table.

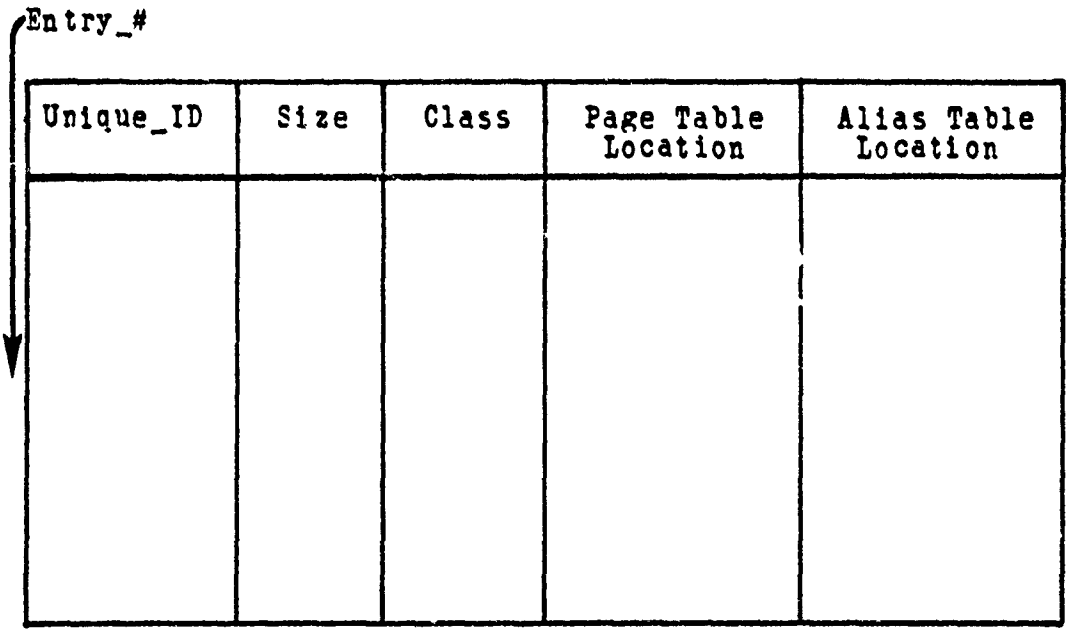


Figure 13. Alias Table.

entry number into the mentor's alias table. When a segment is created, an entry must be made in its mentor segment's alias table. Thus the mentor segment must be known before that segment can be created.

The alias table consists of a header and an array structure of entries. The header has two "pointers" (viz., disk addresses), one that links the alias table to its associated segment and one that links the alias table to the mentor segment's alias table. The header is provided to support the re-construction of the file system after a system crash due to device I/O errors. It is not used at all during normal operations. Each entry in the array structure consists of five fields for identifying the created segments. The Unique_Id field contains the unique identification number for the segment. The Size field is used to record the size of the segment. The Class field contains the appropriate security access class of the segment. The Page_Table_Location field has the disk address of the page table. A null entry indicates a zero-length segment. The Alias_Table_Location field has the disk address of the alias table for the segment. A null entry indicates that the segment is a leaf segment.

4. Memory Management Unit Image

The Memory Management Unit Image (MMU_Image) is a processor local data base. It is an array structure that is

indexed by the DBR_#. Each MMU_Image (see figure 14) includes a software representation of the segment descriptor registers (SDR) for the hardware MMU [12]. This is in exactly the format used by the special I/O instructions for loading/unloading the MMU hardware. The SDR contains the, Base_Address, Limit and Attribute fields for each loaded segment in the process' address space. The Base_Address field contains the base address of the segments in memory (local or global). The Limit field is the number of blocks of contiguous storage for each segment (zero indicates one block). The Attribute field contains eight flags. Five flags are used for protecting the segment against certain types of access. two encode the type of accesses made to the segment (read/write), and one indicates the special structure of the segment [12]. Five of the eight flags in the attribute field are used by the memory manager. The "system only" and "execute only" flags are used to protect the code of the kernel from malicious or unintentional modifications. The "read only" flag is used to control the read or write access to a segment. The "change" flag is used to indicate that the segment has been written into, and the "CPU-inhibit" flag is used to indicate that the segment is not in memory.

The last two fields of the MMU_Image are the Block_Used field and the Maximum_Available_Blocks field. These two fields are used in the mangement of each process' virtual core and are not associated with the hardware MMU.

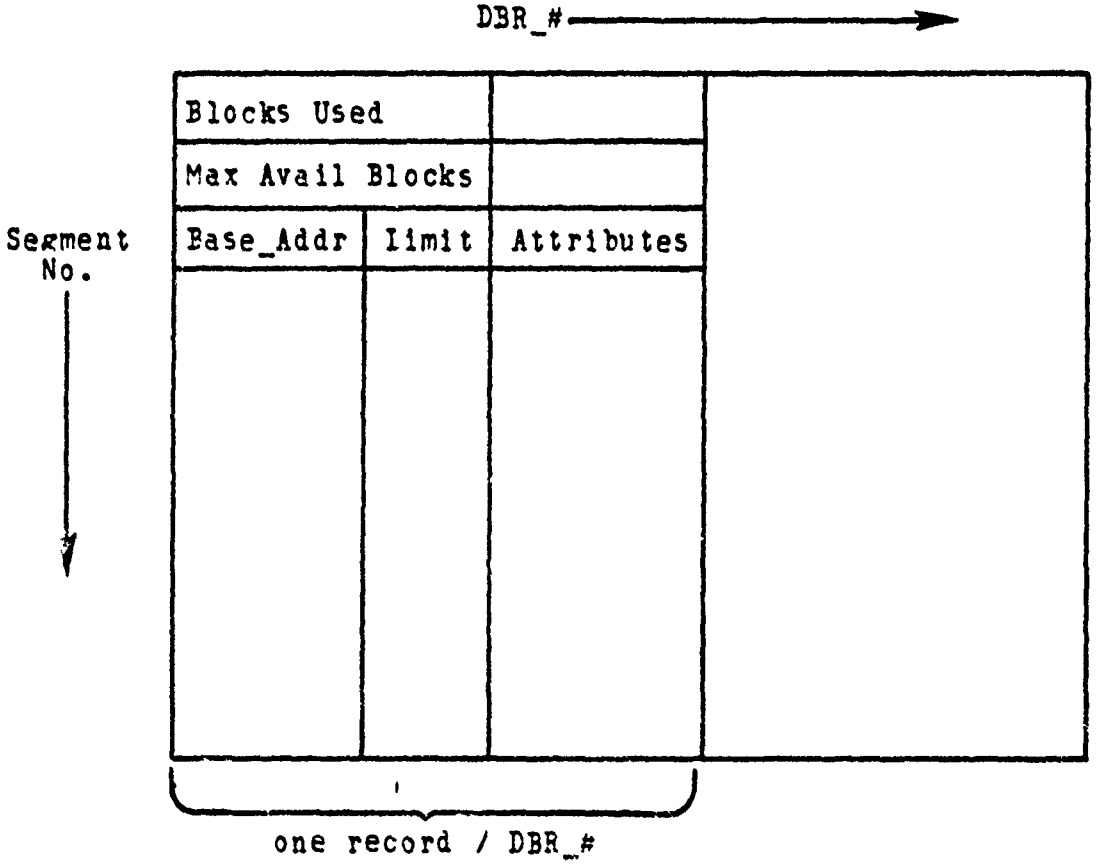


Figure 14. Memory Management Unit Image

5. Memory Allocation/Deallocation Bit Maps

All of the memory allocation/deallocation bit maps (see figure 15) are basically the same structure. Secondary storage, global memory and local memory are managed by memory bit maps. The Disk_Bit_Map is a global resource that is protected from race conditions via the locking convention for the G_AST. Each bit in the bit map is associated with a block of secondary storage. A zero indicates a free block of storage while a one indicates an allocated block of storage. The Global_Memory_Bit_Map is used to manage global memory. It is a shared resource that is protected from race conditions by the locking of the G_AST. The Local_Memory_Bit_Map is the same structure as the Global_Memory_Bit_Map and is used to manage local memory. The Local_Memory_Bit_Map is not locked since it is not a shared resource between memory managers.

D. BASIC FUNCTIONS

The detailed source code for the basic functions and main line of the memory manager are presented in appendices A and B. Appendix A lists the procedures which are coded in PIZ/SYS, while Appendix B lists the lower level hardware dependent procedures which are coded in PIZ/ASM.

PIZ/SYS is a high level modular structured language which produces a machine-independent Z-code similar to

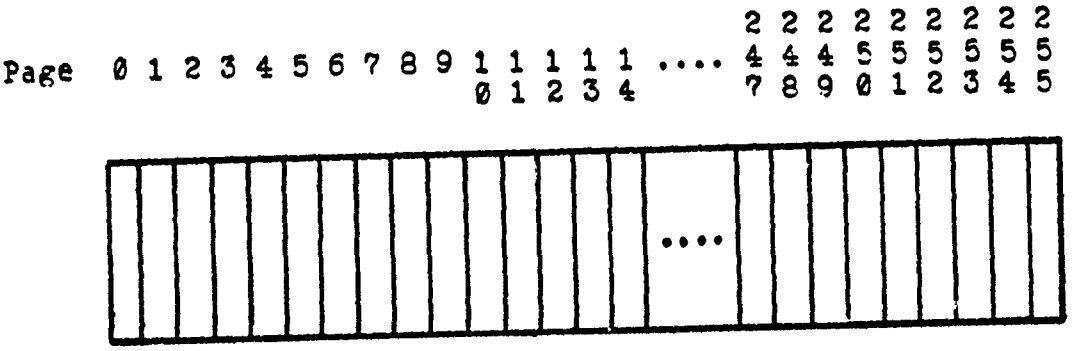


Figure 15. Memory Allocation/Deallocation Map.

PASCAL'S P-code. The translator from Z-code to Z-8000 machine code is currently under development at ZILOG Inc., thus the PLZ/SYS module could not be compiled on the Z8000 [13]. PLZ/ASM is a symbolic assembly language that is used to program the Z-8000. The assembler supports Structured programming and produces a relocatable Z-8000 object module.

In the discussion of the memory manager design, a pseudo-code similar to PLZ/SYS is utilized. The rationale for using this pseudo-code was to provide a summary of the memory manager source code, and to facilitate the presentation of this design.

It is assumed that the memory manager is initialized into the ready state at system generation (as previously mentioned). When the memory manager is initially placed into the running state, it will block itself (via a call to the kernel primitive Wait). Wait will return a message from a signalling process. This message is interpreted by the memory manager to determine the requested function and its required arguments. The function code is used to enter a case statement, which directs the request to the appropriate memory manager procedure.

When the requested action is completed, the memory manager returns a success code (and any additional required data) to the signalling process via a call to the kernel primitive Signal. This call will awaken the process which requested the action to be taken, and place the returned

message into that process' message queue. When that action is completed, the memory manager will return to the top of the loop structure and block itself to wait for the the next request. The main line pseudo-code of the memory manager process is displayed in figure 16.

1. Create an Alias Table Entry

Create_Entry is invoked when a user desires to create a segment. A segment is created by allocating secondary storage, and by making an entry (unique_id, secondary storage location, size, classification) into it's mentor segment's alias table. This implies that the mentor segment must have an alias table associated with it, and that the mentor segment must be active in order to obtain the secondary storage location of the alias table.

The mentor segment can be in one of two states. It may have children (viz., have an alias table), or it may be a leaf segment (viz., not have an alias table). If the mentor segment has children, it has an alias table and this alias table can be read into core, secondary storage can be allocated, and the data can be entered into the alias table. If the mentor segment is a leaf, an alias table must be created for that segment before it (the alias table) can be read into core and data entered into it (see figure 11).

The pseudo-code for CREATE_ENTRY PROCEDURE is presented in figure 17. The arguments passed to Create_Entry

```
ENTRY
    INITIALIZE_PROCESSOR_LOCAL_VARIABLES
    DO
        !  CHECK_IF_MSG_QUEUE_EMPTY  !
        VP_ID, MSG := WAIT
        FUNCTION, ARGUMENTS := VALIDATE_MSG (MSG)
        IF  FUNCTION
            CASE  CREATE_ENTRY  THEN
                SUCCESS_CODE := CREATE_ENTRY (ARGUMENTS)
            CASE  DELETE_ENTRY  THEN
                SUCCESS_CODE := DELETE_ENTRY (ARGUMENTS)
            CASE  ACTIVATE  THEN
                SUCCESS_CODE := ACTIVATE (ARGUMENTS)
            CASE  DEACTIVATE  THEN
                SUCCESS_CODE := DEACTIVATE (ARGUMENTS)
            CASE  SWAP_IN  THEN
                SUCCESS_CODE := SWAP_IN (ARGUMENTS)
            CASE  SWAP_OUT THEN
                SUCCESS_CODE := SWAP_OUT (ARGUMENTS)
            CASE  DEACTIVATE_ALL  THEN
                SUCCESS_CODE := DEACTIVATE_ALL (ARGUMENTS)
            CASE  MOVE_TO_GLOBAL  THEN
                SUCCESS_CODE := MOVE_TO_GLOBAL (ARGUMENTS)
            CASE  MOVE_TO_LOCAL  THEN
                SUCCESS_CODE := MOVE_TO_LOCAL (ARGUMENTS)
            CASE  UPDATE  THEN
                SUCCESS_CODE := UPDATE (ARGUMENTS)
        FI
        SIGNAL (VP_ID, SUCCESS_CODE, ARGUMENTS)
    CD
END  MEMORY_MANAGER_PIZ/SYS  MODULE
```

Figure 16. Memory Manager Mainline Code.

```
CREATE_ENTRY PROCEDURE (PAR_INDEX  WORD, ENTRY_#  WORD,
                        SIZE   WORD, CLASS    BYTE)
   RETURNS (SUCCESS_CODE  BYTE)
   LOCAL  BLKS  WORD, PAGE_TABLE_LOC  WORD
   ENTRY
   IF  ALIAS_TABLE_DOES_NOT_EXIST   THEN
       SUCCESS_CODE := CREATE_ALIAS_TABLE
       IF  SUCCESS_CODE <> VALID  THEN  RETURN
       FI
   FI
   BLKS := CALCULATE_NO_BLKS_REQ (SIZE)
   SUCCESS_CODE := READ_ALIAS_TABLE (
                     G_AST[PAR_INDEX].ALIAS_TABLE_LOC)
   IF  SUCCESS_CODE  <>  VALID  THEN  RETURN
   FI
   SUCCESS_CODE := CHECK_DUP_ENTRY  ! in alias table !
   IF  SUCCESS_CODE  <>  VALID   THEN   RETURN
   FI
   SUCCESS_CODE, PAGE_TABLE_LOC := ALLOC_SEC_STORAGE (BLKS)
   IF  SUCCESS_CODE  <>  VALID   THEN   RETURN
   FI
   UPDATE_ALIAS_TABLE(ENTRY_#, SIZE, CLASS, PAGE_TABLE_LOC)
   SUCCESS_CODE := WRITE_ALIAS_TABLE (
                     G_AST[PAR_INDEX].ALIAS_TABLE_LOC)
   IF SUCCESS_CODE <>  VALID   THEN   RETURN
   ELSE  SUCCESS_CODE := SEG_CREATED
   FI

END CREATE_ENTRY
```

Figure 17. Create Entry Pseudo-code.

are the index into the G_AST for the mentor segment, the entry number into its alias table, the size of the segment to be created, and the security access class of that segment. The return parameter is a success code, which would be "seg_created" for a successful segment creation.

When invoked, Create_Entry will determine which state the mentor segment is in (viz., if it has an alias table). If an alias table does not exist for the mentor segment, one is created and the alias table of the mentor segment's parent is updated. The alias table is read into core and a duplicate entry check is made. If no duplicate entry exists, the segment size is converted from bytes to blocks, and the secondary storage is allocated for non-zero sized segments. The appropriate data is entered into the alias table and the alias table is then written back to secondary storage.

2. Delete an Alias Table Entry

Delete_Entry is invoked when a user desires to delete a segment. A segment is deleted by deallocating secondary storage, and by removing the appropriate entry from the alias table of its mentor segment (the reverse logic of Create_Entry). This implies that the mentor segment must be active at the time of deletion. There are three conditions that can be encountered during the deletion of a

segment: the segment to be deleted may be an inactive leaf segment, an active leaf segment, or a mentor segment.

If the segment to be deleted is an inactive leaf segment (viz., has been swapped out of core, and does not have an entry in the G_AST), the secondary storage can be deallocated and the entry deleted from the mentor segment's alias table. If the segment is an active leaf segment, the segment must first be swapped out of core and deactivated before it can be deleted. This entails signalling the memory manager of each processor, in which the segment is active, to swap out and deactivate the segment.

If the segment to be deleted is a mentor segment, an alias table exists for that segment . If the alias table is empty, the secondary storage for the alias table and the segment can be deallocated, and the entry for the deleted segment can be removed from its mentor's alias table. If the alias table contains any entries, the segment cannot be deleted because these entries would be lost. If this condition is encountered a success code of "leaf_segment_exists" is returned to the process which requested to delete the entry. Due to a confinement problem in "upgraded" segments, this Success_code cannot always be passed outside of the kernel. This implies that the segment manager must strictly prohibit deletion of a segment with an access class not equal to that of the process.

The pseudo-code for DELETE_ENTRY_PROCEDURE is presented in figure 18. The parameters that are passed to this procedure are the parent's index into the G_AST and the entry number into the parent's alias table of the segment to be deleted. The alias_table_loc field is checked to determine the state of the mentor segment (either a leaf or a node), and the appropriate action is then taken. A success code is returned to indicate the results of this procedure.

3. Activate a Segment

Activate is invoked when a user desires to make a segment known by adding a segment to his address space. A segment is activated by making an entry into the L_AST for that processor, and the G_AST. The activated segment could be in one of three states; it could have previously been activated by another process and have a current entry in both the G_AST and L_AST, it could have previously been activated by another process on a different processor and have an entry in the G_AST but not the L_AST, or it could be inactive and have an entry in neither the G_AST nor the L_AST.

If the segment to be activated already has entries in both the L_AST and G_AST, these entries need only be updated to indicate that another process has activated the segment. The segment number is entered into the Segment_No/Access_Auth field of the L_AST, and if the

```
DELETE_ENTRY PROCEDURE ( PAR_INDEX  WORD, ENTRY_#  WORD )
   RETURNS (SUCCESS_CODE  BYTE)
   LOCAL  PAR_INDEX  WORD
   ENTRY
 ! Check if the passed mentor segment has an alias table. !
   IF G_AST[PAR_INDEX].ALIAS_TABLE_LOC <> NULL
       SUCCESS_CODE := READ_ALIAS_TABLE (
                       G_AST[PAR_INDEX].ALIAS_TABLE_LOC)
   ELSE
       SUCCESS_CODE := NO_CHILD_TO_DELETE
   FI
   IF  SUCCESS_CODE  <>  VALID    THEN   RETURN
   FI
 ! Determine if segment has children in alias table !
   IF  ALIAS_TABLE_NOT_EMPTY   THEN
       SUCCESS_CODE := LEAF_SEGMENT_EXISTS
       RETURN    ! Deletion will delete children !
   ELSE
! Search G_AST with UNIQUE_ID to verify segment inactive !
       IF  ACTIVE_IN_G_AST   THEN
           ! Check if active in AST !
           IF  ACTIVE_IN_L_AST  THEN
               DEACTIVATE_ALL (G_AST_INDEX, L_AST_INDEX)
           FI
! Check G_AST to verify segment inactive in other L_AST's !
           IF  ACTIVE_IN_OTHER_L_AST  THEN
               SIGNAL_TO_DEACTIVATE_ALL (G_AST_INDEX)
           FI
       FI
       FREE_SEC_STORAGE_OF_SEG_&_ALIAS_IF_EXISTS
       DELETE_ALIAS_TABLE_ENTRY
   FI
   DELETE_ALIAS_TABLE_ENTRY
   SUCCESS_CODE := WRITE_ALIAS_TABLE (
                   G_AST[PAR_INDEX].ALIAS_TABLE_LOC)
   IF  SUCCESS_CODE = VALID  THEN
       SUCCESS_CODE := SEG_DELETED
   FI
END DELETE_ENTRY
```

Figure 18. Delete Entry Pseudo-code.

segment is a leaf, its mentor's No_Active_Dependents field in the G_AST is incremented. In this design, the G_AST is always searched to determine if the segment has been previously activated by another process.

If the segment to be activated has an entry in the G_AST but not the L_AST, an entry must be made in the L_AST and the G_AST must be updated. The L_AST is searched to determine an available index. The segment number is entered into the L_AST, and the index number is entered into the G_AST Processors_L_ASTE_# field. If the segment to be activated is a leaf segment, its mentor's No_Active_Dependents field in the G_AST is incremented.

If the activated segment does not have an entry in either the G_AST or L_AST, an entry must be made in both. The G_AST is searched to find an available index, and the entry is made. The L_AST is then searched to find an available index, and the entry is made. The L_AST index is then entered into the G_AST Processors_L_ASTE_# field. If the activated segment is a leaf, the No_Active_Dependents field of its mentor's G_AST entry is incremented.

The pseudo-code for ACTIVATE PROCEDURE is presented in figure 19. The parameters that are passed are the DBR_# of the signalling process, the mentor segment's index into the G_AST, the alias table entry number, and the segment number of the activated segment. The mentor segment is always checked to determine if it has an associated alias

```
ACTIVATE  PROCEDURE (DBR_#  BYTE, PAR_INDEX  WORD,
                     ENTRY_#  WORD, SEGMENT_NO  BYTE)
    RETURNS (SUCCESS_CODE  BYTE, RET_G_AST_HANDLE  HANDLE,
                     CLASS  BYTE, SIZE  WORD)
    LOCAL   G_INDEX  WORD,  L_INDEX  WORD
    ENTRY
  ! Verify that passed segment is a mentor segment !
    IF G_AST[PAR_INDEX].ALIAS_TABLE_LOC <> 0 THEN
        SUCCESS_CODE := READ_ALIAS_TABLE (
                        G_AST[PAR_INDEX].ALIAS_TABLE_LOC)
    ELSE
        SUCCESS_CODE := ALIAS_DOES_NOT_EXIST
    FI
    IF  SUCCESS_CODE <>  VALID  THEN   RETURN
    FI
  ! Check G_AST to determine if active !
    SUCCESS_CODE,INDEX := SEARCH_G_AST (UNIQUE_ID)
    IF SUCCESS_CODE = FOUND THEN
        IF  SEGMENT_IN_L_AST  THEN
            UPDATE_L_AST (SEGMENT_NO)
        ELSE
            MAKE_L_AST_ENTRY (DBR_#, SEGMENT_NO)
            UPDATE_G_AST (L_INDEX)
            IF G_AST[INDEX].ALIAS_TABLE_LOC = NULL THEN
                G_AST[PAR_INDEX].NO_DEPENDENTS_ACTIVE += 1
            FI
        FI
    ELSE
        MAKE_G_AST_ENTRY (ENTRY_#)
        MAKE_L_AST_ENTRY (PAR_INDEX, ENTRY_#)
    FI
    SUCCESS_CODE := SEG_ACTIVATED
END ACTIVATE
```

Figure 19. Activate Pseudo-code.

table. If it does not, the success code of "alias_does_not_exist" is returned. If the alias table does exist, it is read into core and the entry number is used as an index to obtain the activated segment's unique_id. The G_AST is then searched to determine if the segment has already been activated. If the unique_id is found, the G_AST is updated and the L_AST is either updated or an entry is made (depending on whether an entry existed or not). If the unique_id of the segment was not found during the search of the G_AST, an entry must be made in both the G_AST and L_AST. Activate returns the activated segment's classification, size, and handle to the signalling process.

4. Deactivate a Segment

Deactivate is invoked when a user desires to remove a segment from his address space. To deactivate a segment, the memory manager either removes or updates an entry in both the L_AST and G_AST. Deactivate uses the reverse logic of activate. Once a segment is deactivated, it can only be reactivated via its mentor's alias table as discussed in activate. If a process requests to deactivate a segment which has not been swapped out of the process' virtual core, the memory manager swaps the segment out and updates the MMU image before the segment is deactivated. The segment to be deactivated could be in one of three states; more than one process could concurrently hold the segment active in the

L_AST, the segment could be held active by one process in the I_AST and more than one in the G_AST, the segment could be held active by only one process in both the L_AST and the G_AST.

Deactivation of leaf segments and mentor segments are handled differently. If the segment is a mentor segment and has active dependents, it cannot be removed from the G_AST (even though no process currently has that segment active). This is based on the design decision which requires that the mentor of all active leaf segments remain in the G_AST to allow access to its alias table. The mentor's alias table must be accessible when an alias table is created for a dependent leaf segment. If a leaf segment is deactivated, the No_Active_Dependents field of its mentor's G_AST entry is decremented. A mentor segment can only be removed from the G_AST if no process holds it active, and it has no active dependents.

If more than one process concurrently hold a segment active in the L_AST, and one of them signals to deactivate that segment, the entry in the L_AST is updated. This is accomplished by nulling out the Segment_No/Access_Auth field of the L_AST for the appropriate process. If required, the No_Active_Dependents field of its mentor segment's G_AST entry is decremented.

If only one process holds the segment active in the L_AST, and that Process signals to deactivate the segment, the L_AST entry for that segment is removed. The Processors_I_ASTE_# is updated and checked to determine if there are other connected processors. If there are no other connected processors and the segment has no active dependents, the segment is removed from the G_AST. If there are other connected processors, the G_AST is updated. If the deactivated segment is a leaf, the mentor segment's No_Active_Dependents field in the G_AST is decremented.

The pseudo-code for DEACTIVATE PROCEDURE is presented in figure 20. The parameters that are passed to the memory manager are the DBR_# of the signalling process, and the index into the G_AST for the segment to be deactivated. The procedure first updates the L_AST, and then removes the entry if no local process holds the segment active. The G_AST is then updated, and its mentor segment is checked (if the deactivated segment was a leaf), to determine if it can be removed. If no processes currently hold the segment active, and it has no active dependents, the segment is removed from the G_AST.

5. Swap a Segment In

SWAP_IN is invoked when a user desires to swap a segment into main memory (global or local) from secondary storage. A segment is swapped into main memory by obtaining

```
DEACTIVATE  PROCEDURE (DBR_#  BYTE, PAR_INDEX  WORD)
    RETURNS (SUCCESS_CODE  BYTE)
    LOCAL   INDEX   WORD
    ENTRY
  ! Check if segment is in core !
    IF G_AST[INDEX].NO_ACTIVE_IN_MEMORY <> Ø THEN
       ! Check MMU image to determine if in local memory !
        IF  IN_LOCAL_MEMORY  THEN
            SUCCESS_CODE := OUT (DBR_#, INDEX)
        FI
    FI
  ! Remove process segment_no entry in L_AST !
    L_AST[L_INDEX].SEGMENT_NO/ACCESS_AUTH[DBR_#] = Ø
    CHECK_IF_ACTIVE_IN_L_AST (L_AST_INDEX)
    IF  NOT_ACTIVE_IN_L_AST  THEN
        L_AST[L_INDEX].MEMORY_ADDR := AVAILABLE
    FI
  ! Check if deleted segment was a leaf !
    IF G_AST[INDEX].G_ASTE_#_PAR <> Ø  THEN
        G_AST[PAR_INDEX].NO_DEPENDENTS_ACTIVE -= 1
  ! Determine if parent can be removed !
        CHECK_FOR_REMOVAL (PAR_INDEX)
    FI
  ! Determine if deactivated segment can be removed !
    CHECK_FOR_REMOVAL (INDEX)
    SUCCESS_CODE  = SEG_DEACTIVATED
END DEACTIVATE
```

Figure 20. Deactivate Pseudo-code.

the secondary storage location of its page table from the G_AST, allocating the required amount of main memory, and reading the segment into the allocated main memory. The segment must be active before it can be swapped into core, and the required main memory space must be available. Three conditions can be encountered during the invocation of SWAP_IN. The segment can already be located in global memory, the segment can already be located in one or more local memories, or the segment may only reside in secondary storage.

If the segment is not in local or global memory, local memory is allocated, the segment is read into the allocated memory, and the appropriate entries are made in the MMU image, the L_AST and the G_AST. If the segment is already in global memory, it can be assumed that the segment is shared and writable. In this case the only required actions are to update the G_AST and L_AST. The No_Active_In_Memory field of the G_AST entry is incremented, and the MMU image is updated to reflect the swapped in segment's core address and attributes.

If the segment already resides in one or more local memories, it must be determined if the segment is "shared" and "writable". A segment is "shared" if it exists in more than one local memory. A segment is "writable" if one process has write access to that segment. If the segment is not shared or not writable and in local memory, the

appropriate entries are updated in the MMU image, the L_AST, and the G_AST. If the segment does not reside in local memory, the required amount of local memory is allocated, the segment is read into the allocated memory, and the appropriate entries are made in the MMU image, the L_AST, and the G_AST.

If the segment is shared, writable, and in local memory, the segment must be moved to global memory. If the segment is not in the memory manager's local memory, it signals another memory manager to move the segment to global memory. After the segment is moved to global memory, the memory manager signals all of the connected memory manager's to update their L_AST and MMU data bases. When all local data bases are current, the memory manager updates the G_AST and returns a success code of seg_activated.

The pseudo-code for SWAP_IN PROCEDURE is presented in figure 21. The arguments passed to SWAP_IN are the G_AST_INDEX of the segment to be moved in, the process' DBR_#, and the access authorized. SWAP_IN will convert the segment size from bytes to blocks, and verify that the process' core will not be exceeded. If the virtual core will be exceeded, a success code of "core_space_exceeded" will be returned. If write access is permitted, the writable bit is set. Checks are then performed to determine the segment's storage location (local or global), and the appropriate action is taken.

```
SWAP_IN  PROCEDURE (INDEX  WORD, DBR_#  BYTE,
                    ACCESS_AUTH  BYTE)
    RETURNS (SUCCESS_CODE  BYTE)
    LOCAL  L_INDEX  WORD,  BLKS  WORD
    ENTRY
    BLKS := CALCULATE_NO._OF_BLKS (G_AST[INDEX].SIZE)
    SUCCESS_CODE := CHECK_MAX_LINEAR_CORE (BLKS)
    IF  SUCCESS_CODE = VIRTUAL_LINEAR_CORE_FULL  THEN
        RETURN
    FI
    G_AST[INDEX].NO_SEGMENTS_IN_MEMORY  +=  1
    IF   ACCESS_AUTH = WRITE  THEN
        G_AST[INDEX].FLAG_BITS := WRITABLE_BIT_SET
    FI
  ! Determine if segment can be put in local memory !
    IF G_AST[INDEX].FLAG_BITS  AND  WRITABLE_MASK = 0
    ORIF  G_AST[INDEX].NO_ACTIVE_IN_MEMORY <= 1  THEN
      ! Determine if already in local memory !
        CHECK_LOCAL_MEMORY (L_AST_INDEX)
        IF  NOT_IN_LOCAL_MEMORY  THEN
            ALLOCATE_LOCAL_MEMORY  (BLKS)
            READ_SEGMENT  (PAGE_TABLE_LOC, BASE_ADDR)
            L_AST[L_INDEX] := BASE_ADDR
        FI
    ELSE
        IF  NOT_IN_GLOBAL_MEMORY  THEN
            UPDATE_MMU
            UPDATE_L_AST
            RETURN
        ELSE
            ALLOCATE_GLOBAL_MEMORY (BLKS)
            IF  IN_LOCAL_MEMORY  THEN
              MOVE_TO_GLOBAL (L_INDEX, BASE_ADDR, SIZE)
            ELSE
              SIGNAL_OTHER_MEMORY_MANAGERS(INDEX,BASE_ADDR)
            FI
        FI
    FI
    UPDATE_MMU_IMAGE (DBR_#,SEG_#,BASE_ADDR,ACCESS,BLKS)
    UPDATE_L_AST_ACCESS (L_INDEX,ACCESS,DBR_#)
    SUCCESS_CODE := SWAPPED_IN
END    SWAP_IN
```

Figure 21. Swap_In Pseudo-code.

6. Swap a Segment Out

SWAP_OUT is invoked when a user desires to move a segment out of core. A segment is swapped out of core by obtaining its secondary storage location, writing the segment to that location (if required), and deallocating the main memory used. The decision to write the segment is determined by the G_AST written bit. This bit is set whenever the segment has been modified. The segment to be swapped out can be in one of two states: the segment can be in local memory, or the segment can be in global memory.

If one process has the segment in local memory and the written bit is set, the segment is written into secondary storage and the local memory is deallocated. If the written bit is not set, the local memory need only be deallocated. If more than one process has the segment in the same local memory, the segment remains in core. The appropriate MMU image is updated to reflect the segments deletion and the G_AST No_Active_In_Memory field is decremented.

All segments in global memory are shared and writable. If a process requests the segment to be swapped out, the segment remains in memory. The MMU image is updated to reflect the segment's deletion, and the G_AST No_Active_In_Memory field is decremented. If the No_Active_In_Memory indicates that one process has the

segment in core, its memory manager is signalled to move the segment to local memory.

The pseudo-code for SWAP_OUT PROCEDURE is presented in figure 22. The arguments passed to SWAP_OUT are the DBR_# of the signalling process, and the G_AST_INDEX of the segment to be removed. The return parameter is a success code. SWAP_OUT removes the segment from the process's virtual core, deletes the segment from its MMU image, and decrements the No_Active_In_Memory field. If the segment can be removed from memory, it is determined which memory can be deallocated. If the segment has been modified, it is written back to secondary storage and the appropriate memory deallocated. If the segment has not been modified, the appropriate memory is deallocated. If after the deletion one process has the segment in global memory, its memory manager need only be signalled to move the segment to local memory. When SWAP_OUT successfully completes, it returns a success code of "swapped out".

7. Deactivate All Segments

DEACTIVATE_ALL is invoked when it becomes necessary to remove a segment from every process' address space. Each process is checked to determine if the segment is active. If a process has the segment active, it is deactivated from its address space. The pseudo code for Deactivate_all is illistrated in figure 23. The parameters passed to

```
SWAP_OUT  PROCEDURE (DBR_#  BYTE, INDEX  WORD)
    RETURNS (SUCCESS_CODE  BYTE)
    ENTRY
    BLKS := G_AST[INDEX].SIZE / BLK_SIZE
    FREE_PROCESS_LINEAR_CORE (BLKS)
    DELETE_MMU_ENTRY (DBR_#, SEG_#)
    G_AST[INDEX].NO_SEGMENTS_IN_MEMORY  -=  1
  ! Determine if segment has been written into !
    IF MMU_IMAGE[DBR_#].SDR[SEG_#].ATTRIBUTES=WRITTEN THEN
      ! If segment has been written into, update G_AST !
        G_AST[INDEX].FLAG_BITS := WRITTEN
    FI
  ! Determine if segment is in global memory !
    IF G_AST[INDEX].GLOBAL_ADDR <> NULL THEN
        IF  G_AST[INDEX].NO_SEGMENTS_IN_MEMORY = 0
        ANDIF  G_AST[INDEX].FLAG_BITS = WRITTEN  THEN
            WRITE_SEG (PAGE_TABLE_LOC, MEMORY_ADDR)
            FREE_LOCAL_BIT_MAP (MEMORY_ADDR,BLKS)
        ELSE
            IF G_AST[INDEX].NO_ACTIVE_IN_MEMORY = 0  THEN
                FREE_LOCAL_BIT_MAP (MEMORY_ADDR,BLKS)
            FI
        FI
    ELSE    ! If not in global memory !
        IF G_AST[INDEX].NO_ACTIVE_IN_MEMORY = 0
        ANDIF G_AST[INDEX].FLAG_BITS = WRITTEN  THEN
            WRITE_SEG (PAGE_TABLE_LOC, GLOBAL_ADDR)
            FREE_GLOBAL_BIT_MAP (GLOBAL_ADDR, BLKS)
        ELSE
            IF G_AST[INDEX].NO_ACTIVE_IN_MEMORY = 0  THEN
                FREE_GLOBAL_BIT_MAP (GLOBAL_ADDR, BLKS)
            FI
        FI
    FI
    SUCCESS_CODE := SWAPPED_OUT
END    SWAP_OUT
```

Figure 22. Swap_Out Pseudo-code.

```
DEACTIVATE_ALL PROCEDURE (INDEX  WORD, L_INDEX  WORD)
    RETURNS (SUCCESS_CODE  BYTE)
    ENTRY
    LOCAL  I  BYTE
       I := 0
       DO
          IF  I = MAX_DBR_#  THEN
            EXIT
          FI
          IF  L_AST[L_INDEX].SEGMENT_NO/ACCESS_AUTH[I]
                <> ZERO  THEN
             SUCCESS_CODE := DEACTIVATE (I, INDEX)
             IF  SUCCESS_CODE <> SEG_DEACTIVATED  THEN
                RETURN
             FI
          FI
          I += 1
       OD
       SUCCESS_CODE := VALID
END DEACTIVATE_ALL
```

Figure 23. Deactivate All Pseudo-code.

Deactivate_all are the deactivated segment's G_AST index and the L_AST index. The I_AST is searched by DBR_# to determine which process has the segment active. If the check reveals that the segment is active, it is deactivated by calling Deactivate. If the segment was successfully deactivated from all processes, a success_code of valid is returned.

8. Move a Segment to Global Memory

MOVE_TO_GLOBAL is invoked when it becomes necessary to move a segment from local to global memory. If a segment resides in one or more local memories, and a process with write access swaps that segment into core, or if a segment resides in in local memory (with write access) and another process with read access requests the segment swapped in, the segment is moved from a local to global memory to avoid a secondary storage access. If the segment resides in the running memory manager's local memory, it will affect the segment transfer, otherwise it will signal another memory manager of a connected processor to affect the transfer. Figure 24 illistrates the pseudo-code for MOVE_TO_GLOBAL. Once the segment has been moved to global memory, the signalled memory manager will update the MMU images for all connected processes, and deallocate the freed local memory. A success code of completed will be returned to the signalling memory manager. The parameters passed to the memory manager are the segment's L_AST index , the global

```
MOVE_TO_GLOBAL PROCEDURE (L_INDEX  WORD, GLOBAL_ADDR  WORD,
                          SIZE  WORD)
    RETURNS (SUCCESS_CODE  BYTE)
    ENTRY
  ! Move segment from local memory to global memory !
    DO_MEMORY_MOVE (MEMORY_ADDR, GLOBAL_ADDR)
    L_AST[INDEX].MEMORY_ADDR := AVAILABLE
  ! Update the MMU image to reflect new address !
    DO  FOR_ALL_DBR'S
      IF L_AST[L_INDEX].SEGMENT_NO/ACCESS_AUTH <> 0 ANDIF
      MMU_IMAGE[DBR_#].SDR[SEG_#].ATTRIBUTES=IN_LOCAL THEN
        MMU_IMAGE[DBR_#].SDR[SEG_#].BASE_ADDR:=GLOBAL_ADDR
      FI
    OD
    SUCCESS_CODE := VALID
END MOVE_TO_GLOBAL
```

Figure 24. Move To Global Pseudo-code.

memory address of the move, and the size of the segment. This information is passed because the G_AST is locked during this request.

9. Move a Segment to Local Memory

MOVE_TO_LOCAL is invoked when it becomes necessary to move a segment from global to local memory. This occurs when one of two processes which hold a segment in global memory swaps the segment out. The segment is moved from global memory to the local memory of the remaining process. Figure 25 illustrates the pseudo-code for MOVE_TO_LOCAL. The parameters passed to the memory manager are the segment's L_AST index, the global address of the segment, and the size of the segment. The return parameter is a success code. The MMU images of the signalled process are updated after the move has been made, and the global memory is deallocated.

10. Update the MMU Image

UPDATE is invoked following a MOVE_TO_GLOBAL operation. After a segment has been moved from local memory to global memory, it is necessary to signal the memory managers of all connected processors to update their MMU images and L_AST with the current location of the segment. They must also deallocate the moved segment's local memory. Figure 26 illustrates the pseudo-code of UPDATE. The parameters passed to the memory manager are the segment's

```
MOVE_TO_LOCAL PROCEDURE (L_INDEX  WORD, GLOBAL_ADDR  WORD,
                         SIZE  WORD)
    RETURNS  (SUCCESS_CODE  BYTE)
    ENTRY
    BLKS := SIZE / BLK_SIZE
    BASE_ADDRESS := ALLOCATE_LOCAL_MEMORY (BLKS)
  ! Move from global to local memory !
    MEMORY_MOVE (GLOBAL_ADDR, BASE_ADDRESS, SIZE)
    LAST[L_INDEX].MEMORY_ADDR := BASE_ADDRESS
    DO  FOR_ALL_DBR'S
      IF LAST[L_INDEX].SEGMENT_NO/ACCESS_AUTH <> 0 ANDIF
      MMU_IMAGE[DBR_#].SDR[SEG_#].ATTRIBUTES=IN_LOCAL THEN
        MMU_IMAGE[DBR_#].SDR[SEG_#].BASE_ADDR:=BASE_ADDRESS
      FI
    OD
    SUCCESS_CODE := VALID
END  MOVE_TO_LOCAL
```

Figure 25. Move To Local Pseudo-code.

```
UPDATE  PROCEDURE (L_INDEX  WORD, GLOBAL_ADDR  WORD,
                   SIZE  WORD)
    RETURNS (SUCCESS_CODE  BYTE)
    ENTRY
    DO  FOR_ALL_DBR'S
      IF  L_AST[L_INDEX].SEGMENT_NO/ACCESS_AUTH <> 0 ANDIF
      MMU_IMAGE[DBR_#].SDR[SEG_#].ATTRIBUTES=IN_LOCAL THEN
         MMU_IMAGE[DBR_#].SDR[SEG_#].BASE_ADDR :=
                                          GLOBAL_ADDR
      FI
    OD
    BLKS := SIZE / BLK_SIZE
    FREE_LOCAL_BIT_MAP (MEMORY_ADDR,BLKS)
    L_AST[L_INDEX].MEMORY_ADDR := ACTIVE
    SUCCESS_CODE := VALID
END UPDATE
```

Figure 26. Update Pseudo-code.

L_AST index, the new global address for the segment, and the size of the segment. The return parameter is a success code.

E. SUMMARY

In this chapter the detailed design of the memory manager process has been presented. The purpose of the memory manager was outlined, followed by a detailed discussion of the memory manager's data bases. The design presented has identified ten basic functions for the memory manager. The implementation details of these functions are presented in Appendix A. The success codes returned by the memory manager are presented in figure 27.

This design has assumed that the kernel level inter-process synchronization primitives will be Saltzer's signal and wait primitives[15]. This fact dominated the design decision to lock the G_AST in the user's process before it signals the memory manager. In a multi-processor environment, the possibility of a deadly embrace exists if the memory manager processes lock the G_AST. Should follow on work implement eventcounts and sequencers as kernel level synchronization primitives, the locking of the G_AST and memory manager synchronization will need to be readdressed.

```
SYSTEM WIDE

INVALID
SWAPPED_IN
SWAPPED_OUT
SEG_ACTIVATED
SEG_DEACTIVATED
SEG_CREATED
SEG_DELETED
VIRTUAL_CORE_FULL
DUPLICATE_ENTRY
READ_ERROR
WRITE_ERROR
DRIVE_NOT_READY
```

```
KERNEL LOCAL

LEAF_SEGMENT_EXISTS
NO_LEAF_EXISTS
ALIAS_DOES_NOT_EXIST
NO_CHILD_TO_DELETE
G_AST_FULL
L_AST_FULL
LOCAL_MEMORY_FULL
GLOBAL_MEMORY_FULL
SECONDARY_STORAGE_FULL
```

```
MEMORY MANAGER LOCAL

VALID
INVALID
FOUND
NOT_FOUND
IN_LOCAL_MEMORY
NOT_IN_LOCAL_MEMORY
 ! + DISK ERRORS !
```

Figure 27. Success Codes

IV. STATUS OF RESEARCH

A. CONCLUSIONS

The memory manager design utilized state of the art software techniques and hardware devices. The design was developed based upon ZILOG'S Z8001 sixteen bit segmented microprocessor used in conjunction with the Z8010 Memory Management Unit[12]. A microprocessor which supports segmentation is required to provide access control of the stored data. The actual implementation of the selected thread was conducted upon the Z8002 non-segmented microprocessor without the Z8010 MMU.

While information security requires that the microprocessor support segmentation, the memory manager was developed to be configuration independent. The design will support a multi-processor environment, and can be easily implemented upon any microprocessor or secondary storage device. The loop free modular design facilitates any required expansion or modification.

Global bus contention is minimized by the memory manager. Segments are stored in global memory only if they are shared and writable. Secondary storage is accessed only if the segment does not currently reside in global memory or some local memory. The controlled sharing of segments

optimizes main memory usage.

The storage of the alias tables in secondary storage supports the recreation of user file hierarchies following a system crash. The aliasing scheme used to address segments supports system security by not allowing the segment's memory location or unique identification to leave the memory manager.

The design of the distributed kernel was clarified by assigning the MMU image management to the memory manager. The transfer of responsibility for memory allocation and deallocation from the supervisor to the memory manager provides support for dynamic memory management.

In conclusion, the memory manager process will securely manage segments in a multi-processor environment. The process is efficient, and is configuration independent. The primitives provided by the memory manager will support the construction of any desired supervisor/user process built upon the kernel.

B. FOLLOW ON WORK

There are several possible areas in the SASS design that can be looked into for continued research. The complete implementation of the memory manager design (refine and optimize the current PLZ/SYS code) is one possibility. Other possibilities include the implementation of dynamic memory

management, and modifying the interface of the memory manager with the distributed kernel using eventcounts and sequencers for inter-process communication.

The implementation of the supervisor has not been addressed to date. Areas of research include the implmentation of the file manager and input/output processes, and the complete design and implementation of the user-host protocols. The implementation of the gatekeeper, and system initialization are other possible research areas. Dynamic process creation and deletion, and the introduction of multi-level hosts could also prove interesting.

```
MEMORY_MANAGER_PLZ_SYS MODULE

!   * * * *  VERS. 1.0  * * * *  !


CONSTANT
            FALSE                       :=  0
            TRUE                        :=  1
            AVAILABLE                   :=  0 ! AST ENTRY AVAIL. !
            ACTIVE                      :=  1 ! AST ENTRY ACTIVE !
            ZERO                        :=  0
            NULL                        :=  %0000
            NULL_PAGE                   :=  0

    !    SUCCESS CODES     !
            INVALID                     :=  0
            VALID                       :=  1
            FOUND                       :=  2
            NOT_FOUND                   :=  3
            SWAPPED_IN                  :=  4
            SWAPPED_OUT                 :=  5
            SEG_ACTIVATED               :=  6
            SEG_DEACTIVATED             :=  7
            SEG_CREATED                 :=  8
            SEG_DELETED                 :=  9
            LEAF_SEGMENT_EXISTS         := 10
            NO_LEAF_EXISTS              := 11
            G_AST_FULL                  := 12
            L_AST_FULL                  := 13
            IN_LOCAL_MEMORY             := 14
            NOT_IN_LOCAL_MEMORY         := 15
            LOCAL_MEMORY_FULL           := 16
            GLOBAL_MEMORY_FULL          := 17
            VIRTUAL_CORE_FULL           := 18
            DUPLICATE_ENTRY             := 19
            NO_CHILD_TO_DELETE          := 20

     !    ATTRIBUTE MASKS     !
            READ_MASK                   := %(2)11111110
            WRITE_MASK                  := %(2)00000001
            CHANGED_MASK                := %(2)01000000
            IN_MEMORY_MASK              := %(2)00000100
            CLEARED                     := ? ! CLEAR ATTRIBUTES '

     !    AUTHORIZED_ACCESS     !
            READ                        := 0
```

```
            WRITE                   := 1
            EXECUTE                 := 2

        !   G_AST_FLAG_BITS_MASKS   !
            WRITABLE_MASK           := %(2)00000010
            WRITTEN_MASK            := %(2)00000100

        !   DESIGN_PARAMETERS       !
            BLK_SIZE                := 256
            MAX_PAGE_SIZE           := BLK_SIZE / 2
            MAX_MSG_SIZE            :=  16
            G_MEM_SIZE              := ? ! SIZEOF GLOBAL MEM !
            L_MEM_SIZE              := ? !SIZEOF LOCAL MEMORY!
            NO_OF_PROCESSORS        := 1
                    ! MAX NUMBER OF DBR_#'S     !
            MAX_DBR_NO              := 4
                    ! MAX ENTRIES IN G_AST       !
            G_AST_LIMIT             := 100
                    ! MAX ENTRIES IN L_AST       !
            L_AST_LIMIT             := 100
                    ! SIZE OF ALIAS TABLE        !
            MAX_ENTRY_NO            := 32
                    ! # OF SEGMENTS PER PROCESS !
            NO_SEG_DESC_REG         := 64
            FIRST_POSS_FREE_BLOCK   :=  1
        !   PROCESSOR_LOCAL_DATA    !
            PROCESSOR_ID            := 0

TYPE        ADDRESS         WORD

            ALIAS_HEADER    RECORD [ SEG_PAGE_TABLE_LOC   WORD
                                     PAR_ALIAS_TABLE_LOC  WORD ]

            ALIAS           RECORD [ UNIQUE_ID     LONG WORD
                                     SIZE                WORD
                                     CLASS               WORD
                                     PAGE_TABLE_LOC      WORD
                                     ALIAS_TABLE_LOC     WORD ]

            SEG_DESC_REG    RECORD [ BASE_ADDR    ADDRESS
                                     LIMIT        BYTE
                                     ATTRIBUTES   BYTE ]

            MMU             RECORD [ SDR ARRAY[NO_SEG_DESC_REG
                                               SEG_DESC_REG ]
                                     BLKS_USED    WORD
                                     MAX_BLKS     WORD  := ???? ]

            G_AST_REC       RECORD [ UNIQUE_ID1          LONG WORD
                                     GLOBAL_ADDR         ADDRESS
                                     PROCESSORS_L_ASTE_NO ARRAY
```

```
                                    [NO_OF_PROCESSORS  WORD]
                                  FLAG_BITS            BYTE
                                  G_ASTE_NO_PAR        WORD
                                  NO_ACTIVE_IN_MEMORY  WORD
                                  NO_ACTIVE_DEPENDENTS WORD
                                  SIZE1                WORD
                                  PAGE_TABLE_LOC1      WORD
                                  ALIAS_TABLE_LOC1     WORD
                                  SEQUENCER            WORD
                                  INSTANCE1            WORD
                                  INSTANCE2            WORD ]

        I_AST_REC     RECORD [ MEMORY_ADDR          ADDRESS
                               SEGMENT_NO_ACCESS_AUTH
                                 ARRAY [MAX_DBR_NO  BYTE] ]

        HANDLE        RECORD [ UNIQUE_ID2   LONG WORD
                               H_INDEX      WORD ]


!*****************************************************************
*                                                               *
*       VARIABLE DECLARATIONS                                   *
*                                                               *
*****************************************************************!

$SECTION G_DATA

GLOBAL      G_AST             ARRAY [G_AST_LIMIT  G_AST_REC]
            GLOBAL_MEM_BIT_MAP  ARRAY[G_MEMORY_SIZE/16 WORD]

$SECTION I_DATA

            MMU_IMAGE         ARRAY [MAX_DBR_NO  MMU]
            I_AST             ARRAY [I_AST_LIMIT   I_AST_REC]
            ALIAS_TABLE  RECORD [ HEADER     ALIAS_HEADER
                                  ALIAS_ENTRY   ARRAY
                                  [MAX_ENTRY_NO  ALIAS] ]
            LOCAL_MEM_BIT_MAP ARRAY [L_MEM_SIZE/16  WORD]
            DISK_BIT_MAP_BUFF ARRAY [????   BYTE]
            PAGE_TABLE_BUFFER ARRAY [BLK_SIZE    BYTE]
```

```
EXTERNAL

!*****************************************************************
*                                                               *
*   The following procedures are coded in PLZ/ASM and are       *
*   contained in a separate PLZ/ASM module.                     *
*                                                               *
*****************************************************************!

READ_PAGE PROCEDURE (DISK_LOC  WORD , MEMORY_ADDR  ADDRESS)
    RETURNS ( SUCCESS_CODE  BYTE )

READ_SEGMENT PROCEDURE (PAGE_TABLE_LOC  WORD , MEMORY_ADDR
                                                ADDRESS)
    RETURNS ( SUCCESS_CODE  BYTE )

WRITE_PAGE PROCEDURE (DISK_LOC WORD , FROM_ADDR  ADDRESS)
    RETURNS ( SUCCESS_CODE BYTE )

WRITE_SEGMENT PROCEDURE (PAGE_TABLE_LOC WORD , FROM_ADDR
                                               ADDRESS)
    RETURNS ( SUCCESS_CODE  BYTE )

READ_DISK_BIT_MAP  PROCEDURE
      RETURNS ( SUCCESS_CODE  BYTE )

WRITE_DISK_BIT_MAP  PROCEDURE
      RETURNS ( SUCCESS_CODE   BYTE )

SEARCH_DISK_BIT_MAP PROCEDURE (START_SRCH_LOC  WORD)
     RETURNS ( SUCCESS_CODE  BYTE, BLK_LOC   WORD  )

CLEAR_DISK_BIT_MAP  PROCEDURE ( BLK_LOC   WORD )

FREE_GLOBAL_BIT_MAP  PROCEDURE (ADDR  ADDRESS, BLKS  WORD)

FREE_LOCAL_BIT_MAP   PROCEDURE (ADDR  ADDRESS, BLKS  WORD)

ALLOC_LOCAL_MEMORY PROCEDURE (BLKS  WORD)
   RETURNS ( SUCCESS_CODE BYTE , BASE_ADDR  ADDRESS )

ALLOC_GLOBAL_MEMORY  PROCEDURE (BLKS  WORD)
     RETURNS ( SUCCESS_CODE  BYTE, BASE_ADDR  ADDRESS )

GET_UNIQ_ID  PROCEDURE
     RETURNS ( ID   LONG WORD, SUCCESS_CODE  BYTE )

MEMORY_MOVE PROCEDURE (TO ADDRESS, FROM  ADDRESS, SIZE WORD)

VALIDATE_MSG  PROCEDURE (MSG   ARRAY [MAX_MSG_SIZE  BYTE])
   RETURNS ( FUNCTION BYTE, ARGUMENTS  ARRAY  [6 WORD] )
```

```
VALIDATE_WAIT_MSG  PROCEDURE (MSG  ARRAY [MAX_MSG_SIZE  BYTE])
   RETURNS ( SUCCESS  BYTE )


INTERNAL

!*****************************************************************
*                                                                *
*   The READ_ALIAS_TABLE Procedure is called from the            *
*   Create_entry procedure and Delete_entry procedure.           *
*   The procedure will read the requested alias table            *
*   from secondary storage to main memory.                       *
*                                                                *
*****************************************************************!

READ_ALIAS_TABLE PROCEDURE ( ALIAS_DISK_LOC  WORD,
                                        MEMORY_ADDR  ADDRESS )
     RETURNS ( SUCCESS_CODE  BYTE )
     ENTRY
     SUCCESS_CODE := READ_PAGE(ALIAS_DISK_LOC, MEMORY_ADDR)
END  READ_ALIAS_TABLE


!*****************************************************************
*                                                                *
*   The WRITE_ALIAS_TABLE Procedure is called from the           *
*   Create_entry and Delete_entry procedures. The pro-           *
*   cedure will write the appropriate alias table from           *
*   main memory to secondary storage.                            *
*                                                                *
*****************************************************************!

WRITE_ALIAS_TABLE   PROCEDURE ( ALIAS_DISK_LOC  WORD,
                                        MEMORY_ADDR  ADDRESS )
     RETURNS ( SUCCESS_CODE   BYTE )
     ENTRY
     SUCCESS_CODE := WRITE_PAGE(ALIAS_DISK_LOC, MEMORY_ADDR)
END  WRITE_ALIAS_TABLE
```

```
!*****************************************************************
*                                                               *
*   The SEARCH_ALIAS_TABLE Procedure is called from the         *
*   Create_alias_table procedure. The procedure will step       *
*   through the alias table until it matches the passed         *
*   unique_id with a table entry, or the table has been         *
*   exhausted. The procedure returns a success code of          *
*   either found or not_found, and the appropriate index        *
*   into the alias table.                                       *
*                                                               *
*****************************************************************!

SEARCH_ALIAS_TABLE PROCEDURE ( UNIQUE_ID  LONG WORD )
     RETURNS  ( SUCCESS_CODE  BYTE, INDEX  BYTE )
    ENTRY
       INDEX := 0
       SUCCESS_CODE := NOT_FOUND
       DO
           IF INDEX > MAX_ENTRY_NO  THEN    EXIT
           FI
           IF ALIAS_TABLE.ALIAS_ENTRY[INDEX].UNIQUE_ID =
                                          UNIQUE_ID  THEN
               SUCCESS_CODE := FOUND
               EXIT
           FI
           INDEX += 1
        OD
END  SEARCH_ALIAS_TABLE


!*****************************************************************
*                                                               *
*   The UPDATE_MMU_IMAGE Procedure is called from the In        *
*   procedure. The procedure will update the MMU image of       *
*   the appropriate process with the memory location,           *
*   limit, and access authorization for the passed segment      *
*   number.                                                     *
*                                                               *
*****************************************************************!

UPDATE_MMU_IMAGE   PROCEDURE (DBR_NO BYTE, SEGMENT_NO  BYTE,
                   ADDR  ADDRESS, ACCESS  BYTE, LIMIT  BYTE )
    LOCAL  ATTR  BYTE
    ENTRY
    MMU_IMAGE[DBR_NO].SDR[SEGMENT_NO].BASE_ADDR  :=  ADDR
    MMU_IMAGE[DBR_NO].SDR[SEGMENT_NO].LIMIT := LIMIT
    ATTR := MMU_IMAGE[DBR_NO].SDR[SEGMENT_NO].ATTRIBUTES
     !  CLEAR PREVIOUS ACCESS  !
    IF ACCESS = READ ORIF ACCESS = WRITE  THEN
        ATTR := ATTR AND %(2)11111110
```

```
    ELSE      ! EXECUTE ONLY ACCESS !
        ATTR := ATTR AND %(2)11110111
    FI
    MMU_IMAGE[DBR_NO].SDR[SEGMENT_NO].ATTRIBUTES :=
                                              ATTR OR ACCESS
END UPDATE_MMU_IMAGE


!*****************************************************************
*                                                                *
*   The DELETE_MMU_ENTRY Procedure is called from the Out        *
*   procedure. The procedure will null out the MMU image         *
*   of the appropriate process for the passed segment            *
*   number.                                                      *
*                                                                *
*****************************************************************!

DELETE_MMU_ENTRY  PROCEDURE ( DBR_NO BYTE, SEGMENT_NO BYTE )
   ENTRY
   MMU_IMAGE[DBR_NO].SDR[SEGMENT_NO].BASE_ADDR := NULL
   MMU_IMAGE[DBR_NO].SDR[SEGMENT_NO].LIMIT := ZERO
   MMU_IMAGE[DBR_NO].SDR[SEGMENT_NO].ATTRIBUTES := CLEARED
END DELETE_MMU_ENTRY


!*****************************************************************
*                                                                *
*   The FIND_SECONDARY_STORAGE Procedure is called from          *
*   the Alloc_sec_storage procedure. The procedure will          *
*   search the secondary storage bit map to find a con-          *
*   tiguous storage location in secondary storage for the        *
*   required number of blocks passed. The procedure will         *
*   return a success code of either valid or invalid.            *
*                                                                *
*****************************************************************!

 FIND_SEC_STORAGE PROCEDURE ( BLKS WORD )
    RETURNS (SUCCESS_CODE BYTE,TABLE ARRAY [BLK_SIZE WORD])
     LOCAL  INDEX  WORD
              I    WORD
     ENTRY
     SUCCESS_CODE := READ_DISK_BIT_MAP
     IF  SUCCESS_CODE <> VALID  THEN
         RETURN
     FI
     INDEX := FIRST_POSS_FREE_BLK
     I := 0
     DO
        SUCCESS_CODE, INDEX := SEARCH_DISK_BIT_MAP (INDEX)
```

```
      IF SUCCESS_CODE <> VALID  THEN
         DO
            CLEAR_DISK_BIT_MAP ( TABLE[I] )
            IF  I = 0   THEN     EXIT
            FI
            I  -=  1
         OD
         SUCCESS_CODE := SEC_STOR_FULL
         RETURN
      FI
      TABLE [I] := INDEX
      I  += 1
      IF  I = BLKS  THEN  EXIT
      FI
   OD
   SUCCESS_CODE := VALID
END  FIND_SEC_STORAGE


!*****************************************************************
*                                                                *
*   The AllOC_ONE_PAGE Procedure is called from the Create       *
*   alias_table procedure. The procedure will find one           *
*   page of secondary storage for the creation of an alias       *
*   table. This procedure will return a success code of          *
*   either valid or invalid.                                     *
*                                                                *
*****************************************************************!

AllOC_ONE_PAGE  PROCEDURE
    RETURNS   ( SUCCESS_CODE  BYTE, PAGE_LOCATION  WORD )
    LOCAL  TABLE  ARRAY[BLK_SIZE  WORD]
    ENTRY
       SUCCESS_CODE, TABLE := FIND_SEC_STORAGE ( 1 )
       IF   SUCCESS_CODE <> VALID  THEN
          RETURN
       FI
       PAGE_LOCATION:= TABLE[0]
END  AILOC_ONE_PAGE
```

```
!*****************************************************************
*                                                                 *
*   The ALLOC_SEC_STORAGE Procedure is called from the            *
*   Create_entry procedure. The procedure will create a           *
*   page table from the allocated secondary storage, and          *
*   write this page to secondary storage. This procedure          *
*   will return a success code of valid or invalid.               *
*                                                                 *
*****************************************************************!

ALLOC_SEC_STORAGE  PROCEDURE ( BLKS  WORD )
    RETURNS ( PAGE_TABLE_LOC   WORD, SUCCESS_CODE   BYTE )
    LOCAL  TABLE  ARRAY [BLK_SIZE  WORD]
    ENTRY
    SUCCESS_CODE, TABLE := FIND_SEC_STORAGE ( BLKS + 1 )
    IF SUCCESS_CODE <> VALID  THEN
       RETURN
    FI
    PAGE_TABLE_LOC := TABLE [0]
    I := 1
    DO
       PAGE_TABLE_BUFFER [I-1] := TABLE [I]
       IF  I = BLKS  THEN  EXIT
       FI
       I += 1
    OD
    DO
        IF  I = MAX_PAGE_SIZE  THEN
            EXIT
        FI
        PAGE_TABLE_BUFFER [I-1]  :=  NULL_PAGE
        I += 1
    OD
    SUCCESS_CODE := WRITE_PAGE ( PAGE_TABLE_LOC,
                                 #PAGE_TABLE_BUFFER )
END  ALLOC_SEC_STORAGE


!*****************************************************************
*                                                                 *
*   The CREATE_ALIAS_TABLE Procedure is called by the             *
*   Create_entry procedure. The procedure will allocate           *
*   secondary storage for the creation of an alias table          *
*   and update the mentor segment's alias table to reflect        *
*   the created alias table's secondary storage location.         *
*   The procedure returns a success code of either valid          *
*   or invalid.                                                   *
*                                                                 *
*****************************************************************!
```

```
CREATE_ALIAS_TABLE  PROCEDURE ( PAR_INDEX  WORD )
    RETURNS ( SUCCESS_CODE   BYTE )
    LOCAL  PARENT  BYTE
           ALIAS_TABLE_LOC  WORD
           ENTRY_NO  BYTE

    ENTRY
       SUCCESS_CODE , ALIAS_TABLE_LOC := ALLOC_ONE_PAGE
       PARENT := G_AST[PAR_INDEX].G_ASTE_NO_PAR
       SUCCESS_CODE := READ_ALIAS_TABLE(G_AST[PARENT].
                       ALIAS_TABLE_LOC1, #ALIAS_TABLE )
       IF SUCCESS_CODE <> VALID THEN
          RETURN
       FI
       SUCCESS_CODE, ENTRY_NO := SEARCH_ALIAS_TABLE(
                                 G_AST[PAR_INDEX].UNIQUE_ID1 )
       IF  SUCCESS_CODE = NOT_FOUND  THEN
          RETURN
       FI
       ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].ALIAS_TABLE_LOC :=
                                          ALIAS_TABLE_LOC
       G_AST[PAR_INDEX].ALIAS_TABLE_LOC1 := ALIAS_TABLE_LOC
       SUCCESS_CODE := WRITE_ALIAS_TABLE ( ALIAS_TABLE_LOC,
                                           #ALIAS_TABLE )
END  CREATE_ALIAS_TABLE


!*****************************************************************
*                                                                *
*   The CHECK_MAX_VIRTUAL_CORE Procedure is called  *            *
*   by the In procedure. The procedure will verify that          *
*   the addition of the segment requested to be swapped in *     *
*   will not cause the process' allocated virtual core to *      *
*   be exceeded. If the virtual core is not exceeded, a          *
*   success code of valid is returned, otherwise a success *     *
*   code of no_memory is returned.                               *
*                                                                *
*****************************************************************!

CHECK_MAX_VIRTUAL_CORE PROCEDURE ( DBR_NO  BYTE,
                                   BLK_NO_REQ  WORD )
    RETURNS    ( SUCCESS_CODE    BYTE )

    ENTRY
       MMU_IMAGE[DBR_NO].BLKS_USED += BLK_NO_REQ
       IF MMU_IMAGE[DBR_NO].BLKS_USED  >
                  MMU_IMAGE[DBR_NO].MAX_BLKS  THEN
          MMU_IMAGE[DBR_NO].BLKS_USED  -=  BLK_NO_REQ
          SUCCESS_CODE := VIRTUAL_CORE_FULL
       ELSE
```

```
            SUCCESS_CODE := VALID
        FI
END  CHECK_MAX_VIRTUAL_CORE


!*****************************************************************
*                                                                *
*   The FREE_PROCESS_VIRTUAL_CORE Procedure is called from       *
*   the Out procedure. The procedure will subtract the           *
*   size of the segment which has been swapped out from          *
*   the virtual linear core allocated to that process.           *
*                                                                *
*****************************************************************!

FREE_PROCESS_VIRTUAL_CORE  PROCEDURE ( BLK_NO   WORD )
     ENTRY
     MMU_IMAGE [ DER_NO ].BLKS_USED -= BLK_NO
END  FREE_PROCESS_VIRTUAL_CORE


!*****************************************************************
*                                                                *
*   The FREE_SECONDARY_STORAGE Procedure is called from          *
*   the Delete_seg procedure. The procedure will read the        *
*   page table of the segment to be deleted and the              *
*   secondary storage bit map into main memory. The bit          *
*   map will be cleared to reflect the deallocation of           *
*   secondary storage, and the page table location will be       *
*   cleared. The procedure returns a success code of             *
*   valid or invalid.                                            *
*                                                                *
*****************************************************************!

FREE_SEC_STORAGE   PROCEDURE ( PAGE_TABLE_LOC  WORD )
    RETURNS ( SUCCESS_CODE   BYTE )
    LOCAL     I       WORD
              TABLE1  ARRAY [ BLK_SIZE  WORD ]
    ENTRY
    SUCCESS_CODE := READ_PAGE ( PAGE_TABLE_LOC , #TABLE1 )
    IF  SUCCESS_CODE <> VALID THEN
        RETURN
    FI
    SUCCESS_CODE := READ_DISK_BIT_MAP
    IF SUCCESS_CODE <> VALID THEN
        RETURN
    FI
    I := 0
    DO
        IF TABLE1[I] = NULL  ORIF I >= BLK_SIZE  THEN
```

```
            EXIT
         FI
         CLEAR_DISK_BIT_MAP ( TABLE1[I] )
         I += 1
      OD
      CLEAR_DISK_BIT_MAP ( PAGE_TABLE_LOC )
      SUCCESS_CODE := VALID
END   FREE_SEC_STORAGE



!*****************************************************************
*                                                                *
*   The DELETE_SEG Procedure is called from the Delete           *
*   entry procedure. The procedure will free secondary           *
*   storage for the deleted segment, and null out the            *
*   entry in its mentor segment's alias table. The pro-          *
*   cedure returns a success code of either valid or in-         *
*   valid.                                                       *
*                                                                *
*****************************************************************!

DELETE_SEG  PROCEDURE ( ENTRY_NO  WORD )
     RETURNS ( SUCCESS_CODE  BYTE )
    ENTRY
    SUCCESS_CODE := FREE_SEC_STORAGE(
          ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].PAGE_TABLE_LOC)
    IF SUCCESS_CODE <> VALID  THEN
       RETURN
    FI
    IF  ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].ALIAS_TABLE_LOC
                                          <>  NULL  THEN
        CLEAR_DISK_BIT_MAP(
         ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].ALIAS_TABLE_LOC)
    FI
    ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].UNIQUE_ID  :=  NULL
END  DELETE_SEG



!*****************************************************************
*                                                                *
*   The CHECK_IF_ALIAS_EMPTY Procedure is called by the          *
*   Delete_entry procedure. The procedure will search the        *
*   alias table to determine if the table is empty. If the       *
*   alias table is empty, the variable Alias_table_empty         *
*   is set equal to true and returned. If the table is not       *
*   empty, Alias_table_empty is set equal to false.              *
*                                                                *
*****************************************************************!
```

```
CHECK_IF_ALIAS_EMPTY  PROCEDURE
    RETURNS ( ALIAS_TABLE_EMPTY  BYTE )
    LOCAL  I    BYTE
    I := 0
    DO
        IF I = ALIAS_TABLE_LIMIT THEN
            ALIAS_TABLE_EMPTY := TRUE
            EXIT
        ELSE
            IF ALIAS_TABLE.ALIAS_ENTRY[I].UNIQUE_ID<>0 THEN
                ALIAS_TABLE_EMPTY := FALSE
                EXIT
            ELSE
                I += 1
            FI
        FI
    OD
END  CHECK_IF_ALIAS_EMPTY


!*****************************************************************
*                                                                *
*   The CHECK_LOCAL_MEMORY Procedure is called from the In *
*   procedure. The procedure determines if the segment is  *
*   in the processor's local memory by examining the MMU   *
*   image for each connected process. If the segment is in *
*   the local memory, the variable Test is set equal to    *
*   true, otherwise it is set equal to false.              *
*                                                                *
*****************************************************************!

CHECK_LOCAL_MEMORY  PROCEDURE ( INDEX  WORD )
    RETURNS ( TEST  BYTE )
    LOCAL    I       BYTE
             SEG_NO   BYTE
    I := 0
    DO
        IF I = MAX_DBR_NO  THEN
            TEST := NOT_IN_LOCAL_MEMORY
            RETURN
        FI
        SEG_NO := ( LAST[INDEX].SEGMENT_NO_ACCESS_AUTH[I]
                      AND  %(2)01111111 )
        IF SEG_NO <> 0  THEN
            IF  (MMU_IMAGE[I].SDR[SEG_NO].ATTRIBUTES  AND
                    IN_MEMORY_MASK)  <> 0  THEN
                TEST := IN_LOCAL_MEMORY
                RETURN
            FI
        FI
```

```
        I += 1
    OD
END  CHECK_LOCAL_MEMORY



!*****************************************************************
*                                                                *
*   The CHECK_FOR_REMOVAL Procedure is called by the Deact-*
*   ivate procedure. The procedure will determine if the   *
*   segment is active in any L_AST and if it has any active*
*   dependents. If the segment is not active and does not  *
*   have any active dependents, the G_AST entry is removed.*
*                                                                *
*****************************************************************!

CHECK_FOR_REMOVAL  PROCEDURE ( INDEX  WORD )
    LOCAL    I     BYTE
             TEST  BYTE
    ENTRY
    TEST := FALSE
    I  := 0
    DO
        IF I = NO_OF_PROCESSORS  ORIF  TEST = TRUE  THEN
            EXIT
        FI
        IF G_AST[INDEX].PROCESSORS_L_ASTE_NO[I] <> 0  THEN
           TEST = TRUE
        FI
        I += 1
    OD
    IF G_AST[INDEX].NO_ACTIVE_DEPENDENTS=0
      ANDIF  TEST = FALSE    THEN
        G_AST[INDEX].UNIQUE_ID1 := AVAILABLE
    FI
END CHECK_FOR_REMOVAL



!*****************************************************************
*                                                                *
*   The CHECK_IF_OTHERS_ACTIVE Procedure is called by the  *
*   Delete_entry procedure. The procedure will check to    *
*   determine if a segment is active in any L_AST. If the  *
*   segment is active, the variable Others_active is set   *
*   equal to true, otherwise it is set equal to false.     *
*                                                                *
*****************************************************************!

CHECK_IF_OTHERS_ACTIVE  PROCEDURE ( INDEX   WORD )
```

```
    RETURNS ( OTHERS_ACTIVE  BYTE )
    LOCAL   I    BYTE
    ENTRY
    I := 0
    DO
       IF I = NO_OF_PROCESSORS THEN
           OTHERS_ACTIVE := FALSE
           RETURN
       FI
       IF G_AST[INDEX].PROCESSORS_L_ASTE_NO[I] <> 0  THEN
           OTHERS_ACTIVE := TRUE
           RETURN
       FI
       I += 1
    OD
END  CHECK_IF_OTHERS_ACTIVE


!*****************************************************************
*                                                               *
*   The ACTIVE_IN_L_AST Procedure is called by the Deact-       *
*   ivate procedure. The procedure will search the Seg-         *
*   ment_#/Access_auth field of a segment to determine if       *
*   the segment is active in the L_AST. If the segment is       *
*   active, the variable Check will be set equal to True        *
*   and returned.                                               *
*                                                               *
*****************************************************************!

ACTIVE_IN_L_AST  PROCEDURE ( INDEX   WORD )
    RETURNS ( CHECK   BYTE )
    LOCAL    I    BYTE
    ENTRY
    I := 0
    CHECK := FALSE
    DO
        IF  I = MAX_DBR_NO  ORIF CHECK = TRUE  THEN
            RETURN
        FI
        IF L_AST[INDEX].SEGMENT_NO_ACCESS_AUTH <> 0  THEN
            CHECK := TRUE
        FI
        I += 1
    OD
END  ACTIVE_IN_L_AST


!*****************************************************************
*                                                               *
```

```
*   The UPDATE_L_AST_ACCESS Procedure is called by the In  *
*   procedure. The procedure will set the read/write bit   *
*   of the appropriate segment_#/access_auth field of the  *
*   L_AST to a one if the process has write access or to a *
*   zero if the process has read access.                   *
*                                                          *
************************************************************!

UPDATE_L_AST_ACCESS  PROCEDURE(INDEX  WORD,ACCESS_AUTH BYTE,
                               DBR_NO  BYTE )
    LOCAL     SEG_NO      WORD
    ENTRY
    SEG_NO := L_AST[INDEX].SEGMENT_NO_ACCESS_AUTH[DBR_NO]
    IF ACCESS_AUTH = WRITE  THEN
        L_AST[INDEX].SEGMENT_NO_ACCESS_AUTH[DBR_NO] :=
                                SEG_NO  OR  %(2)10000000
    ELSE
        L_AST[INDEX].SEGMENT_NO_ACCESS_AUTH[DBR_NO] :=
                                 SEG_NO  AND  %(2)01111111
    FI
END  UPDATE_L_AST_ACCESS



!************************************************************
*                                                          *
*   The SEARCH_G_AST Procedure is called by the Activate   *
*   procedure. The procedure will search the G_AST to      *
*   determine if a passed segment's unique_id exists in    *
*   the G_AST. If the unique_id is found, a success code   *
*   of found and the G_AST index are returned. If the      *
*   segment is not found, a success code of not_found is   *
*   returned.                                              *
*                                                          *
************************************************************!

SEARCH_G_AST   PROCEDURE (SEG_ID   LONGWORD)

   RETURNS (SUCCESS   BYTE, INDEX   WORD)

   LOCAL   I           WORD

   ENTRY
   I := 0
   ILOOP: DO
      IF I => G_AST_LIMIT THEN
         SUCCESS := NOT_FOUND
         INDEX := NULL
         RETURN
      FI
      IF G_AST[I].UNIQUE_ID1 = SEG_ID THEN
```

```
            SUCCESS := FOUND
            INDEX := I
            RETURN
         FI
         I += 1
      OD

END SEARCH_G_AST


!*****************************************************************
*                                                                *
*   The GET_L_AST_INDEX Procedure is called by the Make_         *
*   L_AST_entry procedure. The procedure will search the         *
*   L_AST from top down until an available index is found.       *
*   If an index is not found, a success_code of I_AST_full       *
*   is returned. If an index is found, the index, and a          *
*   success_code of valid are returned.                          *
*                                                                *
*****************************************************************!

GET_L_AST_NO_INDEX      PROCEDURE
   RETURN ( SUCCESS_CODE  BYTE , L_INDEX    WORD)
   LOCAL   I    WORD
   ENTRY
   SUCCESS_CODE := VALID
   I := 0
   ILOOP: DO
      IF I => I_AST_LIMIT THEN
         SUCCESS_CODE := L_AST_FULL
         RETURN
      FI
      IF I_AST[I].MEMORY_ADDR = AVAILABLE THEN
         L_INDEX := I
         I_AST[I].MEMORY_ADDR := ACTIVE
         RETURN
      FI
      I += 1
   OD
END GET_L_AST_NO_INDEX
```

```
!*****************************************************************
*                                                                 *
*   The GET_G_AST_INDEX Procedure is called from the Make_        *
*   G_AST_entry procedure. The procedure will search the          *
*   G_AST from the top down until an available index is           *
*   found. If an index is not found, a success_code of            *
*   G_AST_full is returned. If an index is found, the index       *
*   and a success_code of valid are returned.                     *
*                                                                 *
*****************************************************************!

GET_G_AST_INDEX    PROCEDURE
   RETURN  ( SUCCESS_CODE  BYTE , INDEX   WORD)
   LOCAL   I   WORD
   ENTRY
   SUCCESS_CODE  :=  VALID
   I := 0
   ILOOP: DO
      IF I => G_AST_LIMIT THEN
         SUCCESS_CODE := G_AST_FULL
         RETURN
      FI
      IF G_AST[I].UNIQUID_ID1 = NULL THEN
         INDEX := I
         RETURN
      FI
      I += 1
   OD
END GET_G_AST_INDEX



!*****************************************************************
*                                                                 *
*   The MAKE_G_AST_ENTRY Procedure is called from the             *
*   Activate procedure. The procedure will obtain an              *
*   index irto the G_AST and enter the appropriate data           *
*   from the alias table. The flag bits are set to not            *
*   written and not writable. The eventcounts and ticket          *
*   fields are set to zero. The processor_L_ASTE_# fields         *
*   are set to null. If the entry is successfully made,           *
*   a success_code of valid will be returned.                     *
*                                                                 *
*****************************************************************!

MAKE_G_AST_ENTRY  PROCEDURE (PAR_INDEX  WORD,ENTRY_NO  WORD)
   RETURNS ( SUCCESS_CODE  BYTE, INDEX  WORD )
   LOCAL      I      WORD

   ENTRY
    SUCCESS_CODE, INDEX := GET_G_AST_ENTRY
```

```
    IF  SUCCESS_CODE = VALID    THEN
       G_AST[INDEX].UNIQUID_ID1 := ALIAS_TABLE.ALIAS_ENTRY[
                                    ENTRY_NO].UNIQUID_ID
       G_AST[INDEX].GLOBAL_ADDR := ACTIVE
       G_AST[INDEX].FLAG_BITS := G_AST[INDEX].FLAG_BITS
                                 AND ( NOT  WRITTEN_MASK )
       G_AST[INDEX].FIAG_BITS := G_AST[INDEX].FLAG_BITS
                                 AND ( NOT WRITABLE_MASK )
       G_AST[INDEX].G_ASTE_NO_PAR := PAR_INDEX
       G_AST[INDEX].NO_ACTIVE_IN_MEMORY := 0
       G_AST[INDEX].NO_ACTIVE_DEPENDENTS := 0
       G_AST[INDEX].SIZE1 := ALIAS_TABLE.ALIAS_ENTRY[
                                 ENTRY_NO ].SIZE
       G_AST[INDEX].PAGE_TABLE_LOC1 :=
           ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].PAGE_TABLE_LOC
       G_AST[INDEX].ALIAS_TABLE_LOC1:=
           ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].ALIAS_TABLE_LOC
       G_AST[INDEX].INSTANCE1 := 0
       G_AST[INDEX].INSTANCE2 := 0
       G_AST[INDEX].SEQUENCER := 0
       I := 0
       ILOOP: DO
          IF I = NO_OF_PROCESSORS THEN
             EXIT
          FI
          G_AST[INDEX].PROCESSORS_L_ASTE_NO[I] := NULL
          I += 1
       OD

       SUCCESS_CODE := VALID
    FI
END MAKE_G_AST_ENTRY


!*****************************************************************
*                                                                *
*   The MAKE_I_AST_ENTRY Procedure is called from the            *
*   activate procedure. The procedure will obtain an             *
*   index into the L_AST and enter the appropriate data.         *
*   The memory_addr field is set to active, the segment_         *
*   #/access_auth fields are initialized to zero, and            *
*   the passed segment number is entered into the ap-            *
*   propriate location. If the entry is successfully             *
*   made, a success_code of valid is returned.                   *
*                                                                *
*****************************************************************!

MAKE_I_AST_ENTRY  PROCEDURE (DBR_NO  BYTE, SEGMENT_NO  WORD)
    RETURNS ( SUCCESS_CODE  BYTE, I_INDEX  WORD )
     LOCAL  I   BYTE
```

```
            SEG_NO    WORD
    ENTRY
     SUCCESS_CODE, L_INDEX := GET_L_AST_INDEX
     IF  SUCCESS_CODE <> VALID  THEN     RETURN
     FI
     L_AST[L_INDEX].MEMORY_ADDR := ACTIVE
     I := 0
     DO
        L_AST[L_INDEX].SEGMENT_NO_ACCESS_AUTH[I] := 0
        I  += 1
        IF  I >= MAX_DBR_NO  THEN    EXIT
        FI
     OD
    L_AST[L_INDEX].SEGMENT_NO_ACCESS_AUTH[DBR_NO]:=SEGMENT_NO
END MAKE_L_AST_ENTRY



!*****************************************************************
*                                                                *
*   The DEACTIVATE_ALL Procedure is called  by the               *
*   Detete_entry procedure and  by the Main_line                 *
*   procedure. The procedure will deactivate the                 *
*   deleted segment from all connected process'                  *
*   address space. The G_AST index and the L_AST                 *
*   index for the deleted segment are passed to the              *
*   procedure. If the segment was successfully                   *
*   deactivated from all connected processes, a                  *
*   success_code of valid is returned.                           *
*                                                                *
*****************************************************************!

DEACTIVATE_ALL  PROCEDURE ( INDEX  WORD, L_INDEX  WORD )
    RETURNS ( SUCCESS_CODE   BYTE )
    LOCAL  I  BYTE
    ENTRY
       I := 0
       DO
           IF I = MAX_DBR_NO   THEN   EXIT
           FI
           IF  L_AST[L_INDEX].SEGMENT_NO_ACCESS_AUTH[I]
                                        <> ZERO  THEN
               SUCCESS_CODE := DEACTIVAE ( I, INDEX )
               IF  SUCCESS_CODE <> SEG_DEACTIVATED  THEN
                  RETURN
               FI
            FI
            I += 1
        OD
        SUCCESS_CODE := VALID
END  DEACTIVATE_ALL
```

```
!*****************************************************************
*                                                                 *
*   The SIGNAL_OTHER_MEMORY_MANAGER Procedure is called           *
*   by  the In procedure. The procedure will signal               *
*   a memory manager to move a segment from its local             *
*   memory to global memory. When the segment is moved            *
*   to global memory the procedure will signal all other          *
*   connected memory managers to update their local               *
*   databases. The global address for the transfer                *
*   is passed. A success_code is returned to indicate             *
*   the success of the operation.                                 *
*                                                                 *
*****************************************************************!

SIGNAL_OTHER_MEMORY_MANAGERS   PROCEDURE (
              SEG_INDEX  WORD, ADDR  WORD )
     RETURNS ( SUCCESS_CODE   BYTE )
     LOCAL
       PROCESSOR_NO     BYTE
       FIRST        BYTE
       L_ENTRY_NO    WORD
       VALID_MSG    BYTE
       MSG          ARRAY [MAX_MSG_SIZE  BYTE]
     ENTRY
       FIRST := TRUE
       PROCESSOR_NO  := 0
       DO
         IF  PROCESSOR_NO = PROCESSOR_ID   THEN
              PROCESSOR_NO  +=  1
         FI
         IF  PROCESSOR_NO  >=  NO_OF_PROCESORS  THEN
             EXIT
         FI
         L_ENTRY_NO := G_AST[SEG_INDEX].PROCESSOR_L_ASTE_NO[
                                          PROCESSOR_ID ]
         IF  L_ENTRY_NO  <>  NULL   THEN
             IF  FIRST =  TRUE    THEN
                 FIRST  :=  FALSE
                 IF  PROCESSOR_NO
                 CASE  0   THEN
                   SIGNAL ( VP_ID, MEMORY_MANAGER_0, MOVE,
                   L_ENTRY_NO, ADDR, G_AST[SEG_INDEX].SIZE
                   VP_ID, MSG := WAIT

         !   ****  CHECK IF VALID  MSG  ***    '
                   VALID_MSG := VALIDATE_WAIT_MESSAGE (MSG)
                 FI
             ELSE
```

```
                    IF  PROCESSOR_NO
                    CASE  0  THEN
                      SIGNAL( VP_ID, MEMORY_MANAGER_0, UPDATE,
                      L_ENTRY_NO, ADDR, G_AST[SEG_INDEX].SIZE
                      VP_ID, MSG := WAIT

          |   *****  CHECK  IF  VALID  MSG   *****  |
                      VALID_MSG := VALIDATE_WAIT_MESSAGE(MSG)
                    FI
                  FI
                FI
                PROCESSOR_NO  +=  1
              OD
              IF VALID_MSG  THEN
                  SUCCESS_CODE := VALID
              ELSE
                  SUCCESS_CODE := INVALID
              FI
END SIGNAL_OTHER_MEMORY_MANAGERS


!*****************************************************************
*                                                                *
*   The CREATE_ENTRY Procedure is called by  the                 *
*   main_line procedure. The procedure will create               *
*   an entry into the alias table and allocate sec-              *
*   ondary storage for the created segment. If the               *
*   alias table does not exist, the procedure will               *
*   create an alias table on secondary storage.                  *
*   A unique_id is assigned to the segment and the               *
*   appropriate data is entered into the table.                  *
*   If the function is successfully completed, a                 *
*   success_code of segment_created is returned.                 *
*                                                                *
*****************************************************************!

CREATE_ENTRY  PROCEDURE ( PAR_INDEX  WORD, ENTRY_NO  WORD,
                SIZE  WORD, CLASS  BYTE )
   RETURNS  ( SUCCESS_CODE  BYTE )

    LOCAL     PAGE_TABLE_LOC    WORD
              BLKS      WORD

   ENTRY
      BLKS :=  SIZE / BLK_SIZE
      IF G_AST[PAR_INDEX].G_ASTE_NO_PAR <> ZERO THEN
          SUCCESS_CODE := CREATE_ALIAS_TABLE( PAR_INDEX )
          IF SUCCESS_CODE <> VALID THEN
             RETURN
          FI
```

```
        FI
        SUCCESS_CODE := READ_ALIAS_TABLE(
              G_AST[PAR_INDEX].ALIAS_TABLE_LOC1, #ALIAS_TABLE)
        IF SUCCESS_CODE <> VALID THEN
            RETURN
        FI
        IF ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].UNIQUID_ID <> 0
        THEN
            SUCCESS_CODE := DUPLICATE_ENTRY
            RETURN
        FI
        PAGE_TABLE_LOC, SUCCESS_CODE := ALLOC_SEC_STORAGE(
                                                    BLKS )
        IF SUCCESS_CODE <> VALID THEN
            RETURN
        FI
        ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].UNIQUE_ID,
                              SUCCESS_CODE := GET_UNIQ_ID
        IF  SUCCESS_CODE <> VALID THEN
             RETURN
        FI
        ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].SIZE := SIZE
        ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].CLASS := CLASS
        ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].PAGE_TABLE_LOC :=
                                          PAGE_TABLE_LOC
        ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].ALIAS_TABLE_LOC := 0
        SUCCESS_CODE := WRITE_ALIAS_TABLE(G_AST[PAR_INDEX].
                        ALIAS_TABLE_LOC, #ALIAS_TABLE )
        IF  SUCCESS_CODE = VALID  THEN
           SUCCESS_CODE := SEG_CREATED
        FI
END  CREATE_ENTRY


!*****************************************************************
*                                                                *
*   The DELETE_ENTRY Procedure is called by  the Main-           *
*   line procedure. The procedure will remove a segment          *
*   from secondary storage by deleting its entry in its          *
*   mentor segment's alias table and deallocating its            *
*   allotted secondary storage. Before the segment is            *
*   deleted, the G_AST is checked to ensure that no other        *
*   process holds the segment active, and that the segment       *
*   is not a mentor segment. If the segment is a mentor          *
*   segment, deletion is not allowed. If the segment is          *
*   active, those processes will be signaled to deactivate       *
*   the procedure. When the segment is deactivated, it           *
*   will be deleted. If the deletion is successful, a            *
*   success_code of seg_deleted will be returned.                *
*                                                                *
```

```
*****************************************************************!

DELETE_ENTRY  PROCEDURE ( PAR_INDEX  WORD , ENTRY_NO  WORD \
  RETURNS ( SUCCESS_CODE  BYTE )
  LOCAL  I_INDEX  WORD
         INDEX    WORD
         I      BYTE
         ALIAS_TABLE_EMPTY    BYTE
         OTHERS_ACTIVE        BYTE
  ENTRY
    IF G_AST[PAR_INDEX].ALIAS_TABLE_LOC1 <> NULL  THEN
       SUCCESS_CODE := READ_ALAIS_TABLE ( G_AST[PAR_INDEX].
                       ALIAS_TABLE_LOC1, #ALIAS_TABLE )
    ELSE
        SUCCESS_CODE := NO_CHILD_TO_DELETE
    FI
    IF SUCCESS_CODE  <>  VALID  THEN
        RETURN
    FI
    ALIAS_TABLE_EMPTY := CHECK_IF_ALIAS_EMPTY
    IF  ALIAS_TABLE_EMPTY = TRUE  THEN
       SUCCESS_CODE, INDEX := SEARCH_G_AST (
              ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].UNIQUE_ID )
       IF  SUCCESS_CODE = FOUND  THEN
         I_INDEX := G_AST[PAR_INDEX].PROCESSORS_LAST_NO[
                    PROCESSOR_ID]
         IF  I_INDEX <> NULL  THEN
             SUCCESS_CODE := DEACTIVATE_ALL(INDEX, I_INDEX)
             IF  SUCCESS_CODE <> VALID  THEN
                RETURN
             FI
          FI
          OTHERS_ACTIVE := CHECK_IF_OTHERS_ACTIVE
          IF  OTHERS_ACTIVE = TRUE   THEN
             SIGNAL_OTHERS_TO_DEACTIVATE_ALL
          FI
       FI
       DELETE_SEG ( ENTRY_NO )
       ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].UNIQUE_ID := 0
       SUCCESS_CODE := WRITE_ALIAS_TABLE ( G_AST[PAR_INDEX].
                       ALIAS_TABLE_LOC1, #ALIAS_TABLE )
       IF  SUCCESS_CODE = VALID  THEN
           SUCCESS_CODE := SEG_DELETED
       FI
    ELSE
        SUCCESS_CODE := DEPENDENTS_EXIST
    FI
END  DELETE_ENTRY
```

```
!*****************************************************************
*                                                                 *
*    The ACTIVATE Procedure is called by the Main_line            *
*    procedure. The purpose of activate is to add a               *
*    segment to the user's address space. The procedure           *
*    is passed the segment_#, the parent's handle, and            *
*    the entry number into the alias table for the                *
*    segment. The procedure returns the size,                     *
*    class., and the handle for the activated segment             *
*    The G_AST is searched to determine if the segment            *
*    is already active. If the segment is active and              *
*    not in the L_AST, an entry is made in the L_AST              *
*    and the G_AST is updated. If the segment is active           *
*    in both the G_AST and the L_AST, the entries are             *
*    updated. If the segment was not active, entries              *
*    are made in both the G_AST and the L_AST.                    *
*    If the operation was successfully completed, a               *
*    success_code of seg_activated is returned.                   *
*                                                                 *
*****************************************************************!

ACTIVATE   PROCEDURE (DBR_NO  BYTE, PAR_INDEX  WORD,
                      ENTRY_NO  WORD, SEGMENT_NO  BYTE )
     RETURNS  ( SUCCESS_CODE  BYTE , G_AST_HANDLE  HANDLE ,
                CLASS  BYTE, SIZE   WORD )
     LOCAL  L_INDEX   WORD
            INDEX       WORD
     ENTRY
      IF G_AST[PAR_INDEX].ALIAS_TABLE_LOC1  <>  ZERO   THEN
         SUCCESS_CODE := READ_ALIAS_TABLE(G_AST[PAR_INDEX].
                         ALIAS_TABLE_LOC1, #ALIAS_TABLE)
      ELSE
          SUCCESS_CODE := NO_LEAF_EXIST
      FI
      IF SUCCESS_CODE <> VALID  THEN
          RETURN
      FI
      SUCCESS_CODE , INDEX := SEARCH_G_AST (
                ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].UNIQUE_ID)
      IF  SUCCESS_CODE  =  FOUND   THEN
         L_INDEX := G_AST[INDEX].PROCESSORS_L_ASTE_NO[
                                        PROCESSOR_ID]
         IF  L_INDEX  <>  NULL   THEN
           L_AST[L_INDEX].SEGMENT_NO_ACCESS_AUTH[DBR_NO] :=
                                        SEGMENT_NO
         ELSE
           SUCCESS_CODE, L_INDEX := MAKE_L_AST_ENTRY (
                                        DBR_NO, SEGMENT_NO )
           IF  SUCCESS_CODE  <>  VALID  THEN
              RETURN
           FI
```

```
            G_AST[INDEX].PROCESSORS_L_ASTE_NO[PROCESSOR_ID]
                                    :=  L_INDEX
         FI
         IF  G_AST[INDEX].ALIAS_TABLE_LOC1 = NULL  THEN
            G_AST[PAR_INDEX].NO_DEPENDENTS_ACTIVE  +=  1
         FI
      ELSE
         SUCCESS_CODE, INDEX := MAKE_G_AST_ENTRY(ENTRY_NO)
         IF  SUCCESS_CODE = G_AST_FULL  THEN
            RETURN
         FI
         SUCCESS_CODE, L_INDEX := MAKE_L_AST_ENTRY (
                                       PAR_INDEX, ENTRY_NO )
         IF  SUCCESS_CODE = L_AST_FULL  THEN
            RETURN
         FI
         G_AST[INDEX].PROCESSORS_L_ASTE_NO[PROCESSOR_ID] :=
                                                L_INDEX
      FI
      SUCCESS_CODE := SEG_ACTIVATED
      SIZE  := ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].SIZE
      CLASS := ALIAS_TABLE.ALIAS_ENTRY[ENTRY_NO].CLASS
      G_AST_HANDLE.UNIQUE_ID2 :=G_AST[INDEX].UNIQUE_ID1
      G_AST_HANDLE.INDEX := INDEX
END  ACTIVATE


!*****************************************************************
*                                                                *
*   The SWAP_OUT Procedure is called by the Main_line            *
*   procedure or the Deactivate procedure.  The                  *
*   procedure will remove a segment from main memory             *
*   and store it on secondary storage. The procedure             *
*   is passed the process' DIR_# and the G_AST index             *
*   for the segment to be swapped out of memory.                 *
*   A success_code is returned to indicate the success           *
*   of the operation. The procedure removes the                  *
*   segment from the process' MMU_Image and if not               *
*   shared, it is returned to secondary storage                  *
*   and memory deallocated. Shared segments remain in            *
*   memory until all processes have swapped the segment          *
*   out of main memory.                                          *
*                                                                *
*****************************************************************!

SWAP_OUT  PROCEDURE ( DIR_NO  BYTE, INDEX  WORD )
    RETURNS ( SUCCESS_CODE  BYTE )
    LOCAL   BIAS  WORD
            L_INDEX  WORD
            SEG_NO   WORD
```

```
ENTRY
BLKS := G_AST[INDEX].SIZE1 / BLK_SIZE
L_INDEX:=G_AST[INDEX].PROCESSOR_L_ASTE_NO[PROCESSOR_ID]
SEG_NO := L_AST[L_INDEX].SEGMENT_NO_ACCESS_AUTH[DPP_NO]
FREE_PROCESS_VIRTUAL_CORE ( BLKS )
DELETE_MMU_ENTRY ( DBR_NO, SEG_NO )
G_AST[INDEX].NO_ACTIVE_IN_MEMORY -= 1
IF (MMU_IMAGE[DBR_NO].SDR[SEG_NO].ATTRIBUTES  AND
                              WRITTEN_MASK) <> 0   THEN
    G_AST[INDEX].FLAG_BITS := G_AST[INDEX].FLAG_BITS OR
                                        WRITTEN_MASK

FI
IF  G_AST[INDEX].GLOBAL_ADDR = NULL  THEN
    IF  G_AST[INDEX].NO_ACTIVE_IN_MEMORY = 0  ANDIF
       (G_AST[INDEX].FLAG_BITS AND WRITTEN_MASK) <> 0
    THEN
       SUCCESS_CODE := WRITE_SEGMENT ( G_AST[INDEX].
                           PAGE_TABLE_LOC, L_AST[L_INDEX].
                           MEMORY_ADDR )
       IF SUCCESS_CODE <> VALID  THEN
          RETURN
       FI
       FREE_LOCAL_BIT_MAP ( L_AST[L_INDEX].MEMORY_ADDR,
                                              BLKS )

    ELSE
       IF  G_AST[INDEX].NO_ACTIVE_IN_MEMORY = 0  THEN
           FREE_LOCAL_BIT_MAP ( L_AST[L_INDEX].
                                  MEMORY_ADDR, BLKS )

       FI
    FI
 ELSE
    IF  G_AST[INDEX].NO_ACTIVE_IN_MEMORY = 0  ANDIF
    (G_AST[INDEX].FLAG_BITS AND WRITTEN_MASK) <> 0 THEN
        SUCCESS_CODE := WRITE_SEGMENT ( G_AST[INDEX].
            PAGE_TABLE_LOC1, G_AST[INDEX].GLOBAL_ADDR )
        IF  SUCCESS_CODE <> VALID  THEN
            RETURN
        FI
        FREE_GLOBAL_BIT_MAP ( G_AST[INDEX].GLOBAL_ADDR,
                                              BLKS )

      ELSE
        IF  G_AST[INDEX].NO_ACTIVE_IN_MEMORY = 0  THEN
          FREE_GLOBAL_BIT_MAP( G_AST[INDEX].GLOBAL_ADDR,
                                              BLKS )

        FI
      FI
 FI
 SUCCESS_CODE := SWAPPED_OUT
END SWAP_OUT
```

```
!*****************************************************************
*                                                                 *
*    The DEACTIVATE Procedure  is called by  the                  *
*    the Main_line procedure, the Deactivate_all                  *
*    procedure, or the Delete_entry procedure.                    *
*    The purpose of deactivate is to remove a segment             *
*    from a process' address space. The segment is                *
*    removed by deleting the segment number from the              *
*    L_AST. If no other processes have the segment                *
*    active and no children are active, the entry                 *
*    is removed from the L_AST and the G_AST.                     *
*    The process' DBR_# and the deactivated segment's             *
*    G_AST index are passed to the procedure. A                   *
*    success_code is returned to indicate the success             *
*    of the operation.                                            *
*                                                                 *
*****************************************************************!

DEACTIVATE  PROCEDURE ( DBR_NO  BYTE, INDEX  WORD )
   RETURNS  ( SUCCESS_CODE  BYTE )
   LOCAL    L_INDEX   WORD
            SEG_NO     BYTE
            CHECK      BYTE
            PAR_INDEX  WORD
   ENTRY
   PAR_INDEX := G_AST[INDEX].G_ASTE_NO_PAR
   L_INDEX := G_AST[INDEX].PROCESSOR_L_ASTE_NO[PROCESSOR_ID]
   SEG_NO := L_AST[L_INDEX].SEGMENT_NO_ACCESS_AUTH[DBR_NO]
   IF  G_AST[INDEX].NO_ACTIVE_IN_MEMORY <> 0  THEN
      IF (MMU_IMAGE[DBR_NO].SDR[SEG_NO].ATTRIBUTES AND
                           IN_MEMORY_MASK) = ZERO   THEN
         SUCCESS_CODE := SWAP_OUT ( DBR_NO, INDEX )
         IF  SUCCESS_CODE <> SWAPPED_OUT   THEN
            RETURN
         FI
      FI
   FI
   L_AST[L_INDEX].SEGMENT_NO_ACCESS_AUTH[DBR_NO] := 0
   CHECK := ACTIVE_IN_L_AST( L_INDEX )
   IF  CHECK = 0  THEN
      L_AST[L_INDEX].MEMORY_ADDR := AVAILABLE
   FI
   IF  PAR_INDEX  <> 0  THEN
      G_AST[PAR_INDEX].NO_ACTIVE_DEPENDENTS -= 1
      CHECK_FOR_REMOVAL ( PAR_INDEX )
   FI
   CHECK_FOR_REMOVAL ( INDEX )
   SUCCESS_CODE := SEG_DEACTIVATED
END DEACTIVATE
```

```
!*****************************************************************
*                                                                 *
*    The MOVE_TO_GLOBAL Procedure is called by the                *
*    Main_line procedure. The procedure is called to              *
*    to move a shared and writable segment to global              *
*    memory. The procedure is passed the L_AST index,             *
*    the size, and the global address for the move.               *
*    A success_code is returned to indicate the                   *
*    success of the operation. The procedure locates              *
*    the segment in its local memory, transfers the               *
*    segment to global memory, and deallocates the                *
*    local memory.                                                *
*                                                                 *
*****************************************************************!

MOVE_TO_GLOBAL  PROCEDURE ( L_INDEX  WORD, GLOBAL_ADDR
                            ADDRESS, SIZE   WORD )
    RETURNS ( SUCCESS_CODE  BYTE )
    LOCAL    SEG_NO   BYTE
             I        BYTE
    ENTRY
    MEMORY_MOVE ( L_AST[L_INDEX].MEMORY_ADDR, GLOBAL_ADDR,
                                               SIZE )
    L_AST[L_INDEX].MEMORY_ADDR  :=  ACTIVE
    I := 0
    DO
       IF  I = MAX_DBR_NO  THEN   EXIT
       FI
       SEG_NO := L_AST[L_INDEX].SEGMENT_NO_ACCESS_AUTH[I]
                 AND  %(2)01111111
       IF  SEG_NO <> 0  ANDIF  (MMU_IMAGE[I].SDR[SEG_NO].
                 ATTRIBUTES AND IN_MEMORY_MASK) = 0  THEN
           MMU_IMAGE[I].SDR[SEG_NO].BASE_ADDR := GLOBAL_ADDR
       FI
       I += 1
     OD
     FREE_LOCAL_BIT_MAP ( L_AST[LINDEX].MEMORY_ADDR, BLKS )
     SUCCESS_CODE := VALID
END  MOV_TO_GLOBAL
```

```
!*****************************************************************
*                                                                 *
*   The SWAP_IN Procedure is called by the Main_line              *
*   procedure. The procedure will transfer a segment              *
*   from secondary storage to main memory. The procedure          *
*   is passed the process' DBR_#, the segment's G_AST             *
*   index, and the authorized access to the segment.              *
*   A success_code is returned to indicate the                    *
*   success of the operation. ( successful = swapped_in )         *
*   If the segment is not already in memory, the appro-           *
*   priate memory is allocated and the segment is trans-          *
*   fered to the allocated memory. If the segment is              *
*   writable and shared, the segment is transfered into           *
*   global memory.                                                *
*                                                                 *
*****************************************************************!

SWAP_IN PROCEDURE(INDEX WORD,DBR_NO  BYTE,ACCESS_AUTH  BYTE)
    RETURNS ( SUCCESS_CODE  BYTE )
    LOCAL  BLKS  WORD,
           TEST  BYTE
           SEG_NO  BYTE
           L_INDEX  WORD
           BASE_ADDR  ADDRESS
    ENTRY
    BLKS := G_AST[INDEX].SIZE / BLK_SIZE
    L_INDEX:=G_AST[INDEX].PROCESSOR_L_ASTE_NO[PROCESSOR_ID]
    SEG_NO := L_AST[L_INDEX].SEGMENT_NO_ACCESS_AUTH[DBR_NO]
    SUCCESS_CODE := CHECK_MAX_VIRTUAL_CORE ( DBR_NO, BLKS )
    IF  SUCCESS_CODE = VIRTUAL_CORE_FULL   THEN
       RETURN
    FI
    G_AST[INDEX].NO_ACTIVE_IN_MEMORY  += 1
    IF  ACCESS_AUTH = WRITE  THEN
       G_AST[INDEX].FLAG_BITS := G_AST[INDEX].FLAG_BITS OR
                                              WRITABLE_MASK
    FI
    IF  (G_AST[INDEX].FLAG_BITS AND WRITABLE_MASK) = 0
    OFIF G_AST[INDEX].NO_ACTIVE_IN_MEMORY <= 1  THEN
        TEST := CHECK_LOCAL_MEMORY ( L_INDEX )
        IF  TEST <> IN_LOCAL_MEMORY  THEN
          SUCCESS_CODE,BASE_ADDR := ALLOC_LOCAL_MEMORY(BLKS)
           IF  SUCCESS_CODE = LOCAL_MEMORY_FULL  THEN
              RETURN
           FI
           SUCCESS_CODE := READ_SEGMENT ( G_AST[INDEX].
                                PAGE_TABLE_LOC1, BASE_ADDR )
           IF  SUCCESS_CODE <> VALID  THEN
               FREE_LOCAL_BIT_MAP ( BASE_ADDR, BLKS )
               RETURN
           FI
```

```
            L_AST[L_INDEX].MEMORY_ADDR := BASE_ADDR
        ELSE
            BASE_ADDR := L_AST[L_INDEX].MEMORY_ADDR
        FI
    ELSE
        IF  G_AST[INDEX].GLOBAL_ADDR = NULL  THEN
            SUCCESS_CODE, BASE_ADDR := ALLOC_GLOBAL_MEMORY(
                                                    BLKS )
            IF  SUCCESS_CODE = GLOBAL_MEMORY_FULL  THEN
              RETURN
            FI
            IF  TEST = IN_LOCAL  THEN
              SUCCESS_CODE := MOVE_TO_GLOBAL ( L_INDEX,
                              BASE_ADDR,G_AST[INDEX].SIZE1)
              IF SUCCESS_CODE <> VALID  THEN
                FREE_GLOBAL_BIT_MAP ( BASE_ADDR, BLKS )
                RETURN
              FI
            ELSE
              SUCCESS_CODE :=
              SIGNAL_OTHER_MEMORY_MANAGERS(INDEX,BASE_ADDR)
              IF  SUCCESS_CODE <> VALID  THEN
                RETURN
              FI
            FI
        ELSE
            BASE_ADDR := G_AST[INDEX].GLOBAL_ADDR
        FI
    FI
    UPDATE_MMU_IMAGE(DES_NO, SEG_NO, BASE_ADDR, ACCESS_AUTH,
                                                    BLKS )
    UPDATE_I_AST_ACCESS ( L_INDEX, ACCESS_AUTH, DES_NO )
    SUCCESS_CODE := SWAPPED_IN
END SWAP_IN


!*************************************************************
*                                                            *
*   The MOVE_TO_LOCAL Procedure is called by the Main_       *
*   line procedure. The procedure is called when             *
*   a segment no longer needs to be in global memory         *
*   and can be moved to local memory. The procedure          *
*   is passed the I_AST index, size, and global address      *
*   of the segment to be moved. A success_code is returned   *
*   to indicate the success of the operation.                *
*                                                            *
*************************************************************!

MOVE_TO_LOCAL  PROCEDURE ( L_INDEX  WORD, GLOBAL_ADDR
                           ADDRESS,  SIZE  WORD )
```

```
    RETURNS ( SUCCESS_CODE  BYTE )
    LOCAL   BASE_ADDRESS    ADDRESS
            SEG_NO    BYTE
            I         BYTE
            BLKS      BYTE
     ENTRY
     BLKS := SIZE / BLK_SIZE
     SUCCESS_CODE, BASE_ADDRESS := ALLOC_LOCAL_MEMORY(BLKS)
     IF  SUCCESS_CODE <> VALID  THEN
        RETURN
     FI
     MEMORY_MOVE ( GLOBAL_ADDR, BASE_ADDRESS, SIZE )
     L_AST[L_INDEX].MEMORY_ADDR := BASE_ADDRESS
     I := 0
     DO
        IF  I = MAX_DBR_NO  THEN  EXIT
        FI
        SEG_NO := L_AST[L_INDEX].SEGMENT_NO_ACCESS_AUTH[I]
                  AND  %(2)01111111
         IF SEG_NO <> 0 ANDIF (MMU_IMAGE[I].SDR[SEG_NO].
                  ATTRIBUTES AND IN_MEMORY_MASK) = 0  THEN
           MMU_IMAGE[I].SDR[SEG_NO].BASE_ADDR:=BASE_ADDRESS
         FI
         I += 1
      OD
     SUCCESS_CODE = VALID
END  MOVE_TO_LOCAL


!*****************************************************************
*                                                                *
*   The UPDATE Procedure is called by the Main_line              *
*   procedure. The procedure is called to update the             *
*   MMU images of process' connected to a segment                *
*   that was moved to global memory by the Move_to_global        *
*   procedure. The procedure is passed the L_AST index,          *
*   the size, and the global address of the segment              *
*   that was moved to global address. A success_code             *
*   is returned to indicate the success of the operation.        *
*                                                                *
*****************************************************************!

UPDATE  PROCEDURE ( L_INDEX  WORD, GLOBAL_ADDR  ADDRESS,
                                   SIZE   WORD )
    RETURNS ( SUCCESS_CODE   BYTE )
    LOCAL   SEG_NO    BYTE
            BLKS      BYTE
            I         BYTE
    ENTRY
    I := 0
```

```
    DO
       IF  I = MAX_DBR_NO   THEN    EXIT
       FI
       SEG_NO := L_AST[L_INDEX].SEGMENT_NO_ACCESS_AUTH[I]
                 AND  %(2)01111111
       IF  SEG_NO <> 0  ANDIF (MMU_IMAGE[I].SDR[SEG_NO].
          ATTRIBUTES AND IN_MEMORY_MASK) = 0  THEN
          MMU_IMAGE[I].SDR[SEG_NO].BASE_ADDR := GLOBAL_ADDR
       FI
       I += 1
    OD
    BLKS := SIZE / BLK_SIZE
    FREE_LOCAL_BIT_MAP( L_AST[L_INDEX].MEMORY_ADDR, BLKS )
    L_AST[L_INDEX].MEMORY_ADDR := ACTIVE
    SUCCESS_CODE := VALID
END UPDATE




    !       *****    MAIN LINE  CODE    *****    !

$SECTION MAIN

MAIN_LINE  PROCEDURE
    LOCAL   FUNCTION  BYTE
            ARGUMENTS ARRAY [ ???  BYTE]
            MSG       ARRAY [MAX_MSG_SIZE  BYTE]
            VP_ID     BYTE
            SUCCESS_CODE BYTE
  ENTRY
   INITIALIZE_PROCESSOR_LOCAL_VARIABLES
   DO
      CHECK_MSG_QUEUE
      VP_ID, MSG := WAIT
        !   ***  VALIDATE  THE  MSG  FROM  WAIT  ***    !
        FUNCTION, ARGUMENTS := VALIDATE_MSG ( MSG )
      IF  FUNCTION
        CASE CREATE_ENTRY  THEN  SUCCESS_CODE :=
                                   CREATE_ENTRY(ARGUMENTS)
        CASE DELETE_ENTRY  THEN  SUCCESS_CODE :=
                                   DELETE_ENTRY(ARGUMENTS)
        CASE ACTIVATE THEN SUCCESS_CODE,HANDLE,CLASS,SIZE :=
                                   ACTIVATE(ARGUMENTS)
        CASE DEACTIVATE    THEN  SUCCESS_CODE :=
                                   DEACTIVATE(ARGUMENTS)
        CASE SWAP_IN           THEN  SUCCESS_CODE :=
                                   SWAP_IN(ARGUMENTS)
        CASE SWAP_OUT           THEN  SUCCESS_CODE :=
                                   SWAP_OUT(ARGUMENTS)
```

```
            CASE MOVE_TO_LOCAL THEN SUCCESS_CODE :=
                                    MOVE_TO_LOCAL(ARGUMENTS)
            CASE MOVE_TO_GLOBAL THEN SUCCESS_CODE :=
                                    MOVE_TO_GLOBAL(ARGUMENTS)
            CASE UPDATE         THEN SUCCESS_CODE :=
                                    UPDATE(ARGUMENTS)
            CASE DEACTIVAE_ALL THEN SUCCESS_CODE :=
                                    DEACTIVATE_ALL(ARGUMENTS)
         FI
         SIGNAL ( VP_ID, SUCCESS_CODE, ARGUMENTS )
      OD
END MAIN_LINE
END MEMORY_MANAGER_PLZ_SYS MODULE
```

```
!*****************************************************************!
!  THE PLZ/ASM MODULE WAS WRITTEN TO PROVIDE SUPPORT FOR          !
!  THE SWAP_IN THREAD [APPENDIX 3]. THE VALIDITY OF THE           !
!  CODE HAS NOT BEEN THOROUGHLY TESTED, NOR HAS IT BEEN           !
!  OPTIMIZED. THE CODE SIMULATES SECONDARY STORAGE IN             !
!  MAIN MEMORY, AND WAS NOT INTENDED TO BE USED IN AN             !
!  ACTUAL SYSTEM IMPLEMENTATION.                                  !
!*****************************************************************!

M_MGR_2 MODULE

!   * * * *   VERS. 1.0   * * * *  !


CONSTANT
            FALSE               := 0
            TRUE                := 1
            AVAILABLE           := 0   ! AST ENTRY AVAILABLE !
            ACTIVE              := 1   ! AST ENTRY ACTIVE !
            ZERO                := 0
            NULL                := %0000
            NULL_PAGE           := 0
            HBUG                := %A900
            MONITOR             := %059A

              !    SUCCESS CODES   !
            INVALID             := 0
            VALID               := 1
            FOUND               := 2
            NOT_FOUND           := 3
            SWAPPED_IN          := 4
            SWAPPED_OUT         := 5
            SEG_ACTIVATED       := 6
            SEG_DEACTIVATED     := 7
            SEG_CREATED         := 8
            SEG_DELETED         := 9
            LEAF_SEG_EXISTS     := 10
            NO_LEAF_EXISTS      := 11
            G_AST_FULL          := 12
            L_AST_FULL          := 13
            IN_LOCAL_MEMORY     := 14
            NOT_IN_LOCAL_MEM    := 15
            LOCAL_MEMORY_FULL   := 16
            GLOBAL_MEM_FULL     := 17
            VIRTUAL_CORE_FULL   := 18
            DUPLICATE_ENTRY     := 19
```

```
            NO_CHILD_TO_DEL   := 20
            SEC_STOR_FULL     := 21
            DISK_ERROR        := 22
            ALIAS_DOES_NOT_EXIST  := 23
    !    ATTRIBUTE MASKS    !
            READ_MASK         := %(2)11111110
            WRITE_MASK        := %(2)00000001
            CHANGED_MASK      := %(2)01000000
            IN_MEMORY_MASK    := %(2)00000100
            CLEARED           := 0              ! CLEAR ATTR !
    !    AUTHORIZED ACCESS     !
            READ              := 0
            WRITE             := 1
            EXECUTE           := %(2)00001000

    !   G_AST  FLAG BITS FIELD MASKS   !
            WRITAPLE_MASK     := %(2)00000010
            WRITTEN_MASK      := %(2)00000100

    !   DESIGN  PARAMETERS     !
            BLK_SIZE          := 128
            MAX_PAGE_SIZE     := BLK_SIZE/2
            NO_OF_PROCESSORS  := 1
            MAX_DBR_NO        := 4  ! EVEN NO. OF DBR_#'S !
            G_AST_LIMIT       := 16  ! MAX ENTRIES IN G_AST !
            L_AST_LIMIT       := 16  ! MAX ENTRIES IN L_AST !
            MAX_ENTRY_NO      := 10  ! SIZE OF ALIAS TABLE !
            NO_SEG_DESC_REG   := 8   ! NO. OF SEGMENT/PROCESS!
            FST_POSS_FREE_BLK := 1
            DISK_MEM_BASE     := %9000
            MAX_POSS_D_BLKS   := 96
            GLOBAL_MEM_BASE   := %8000
            MAX_POSS_G_BLKS   := 32
            LOCAL_MEM_BASE    := %6000
            MAX_POSS_L_BLKS   := 64
            DISK_BIT_MAP_LOC  := 0
TYPE
            ADDRESS             WORD
            ALIAS_HEADER        RECORD [
                                  SEG_PAGE_TABLE_LOC  WORD
                                  PAR_ALIAS_TABLE_LOC WORD ]

            SEG_DESC_REG        RECORD [
                                  BASE_ADDR   ADDRESS
                                  LIMIT       BYTE
                                  ATTRIBUTE   BYTE ]

            ALIAS               RECORD [
                                  UNIQUE_ID     WORD
                                  CLASS         WORD
                                  SIZE          WORD
```

```
                                    PAGE_TABLE_LOC WORD
                                    ALIAS_TABLE_LOC WORD ]

          MMU                     RECORD [
                                    SDR ARRAY  [NO_SEG_DESC_REG
                                               SEG_DESC_REG]
                                    BLKS_USED       WORD
                                    MAX_BLKS        WORD]

GLOBAL

  !$SECTION  G_DATA  !

        GLOBAL_MEM_BIT_MAP ARRAY [MAX_POSS_G_BLKS/16 WORD]
        G_AST_LOCK         BYTE

  ! $SECTION L_DATA  !

         MMU_IMAGE          ARRAY [MAX_DBR_NO  MMU]
         LOCAL_MEM_BIT_MAP  ARRAY [MAX_POSS_L_BLKS/16 WORD]
         ALIAS_TABLE        RECORD [ HEADER   ALIAS_HEADER
                                     ALIAS_ENTRY  ARRAY
                                     [MAX_ENTRY_NO  ALIAS] ]
         DISK_BIT_MAP BUFF  ARRAY [6   BYTE]
         PAGE_TABLE_BUFFER  ARRAY [BLK_SIZE   BYTE]
INTERNAL

COMPACT_L  PROCEDURE
    ENTRY
END COMPACT_L


COMPACT_G  PROCEDURE
    ENTRY
END COMPACT_G

GLOBAL

ALLOC_LOCAL_MEMORY  PROCEDURE
    !*****************************************!
    ! PASSED PARAMETER                        !
    !   R0 = BLKS OF MEMORY                   !
    ! RETURNED PARAMETERS                     !
    !   R0 = SUCCESS_CODE                     !
    !   R1 = BASE_ADDR                        !
    ! LOCAL VARIABLES                         !
    !   R0 = BLKS                             !
    !   R10 = BIT_MAP_INDEX                   !
    !   R11 = COUNTER FOR BIT                 !
    !   R12 = BIT_MAP WORD                    !
    !   R13 = WORKING REGISTER                !
```

```
!*****************************************!
  LOCAL   BLKS  WORD
          IS_COMPACTED  BYTE
          FILLER2   BYTE
 ENTRY
 LD  BLKS, R0
 LDB  IS_COMPACTED, #FALSE
 LD  R10, #ZERO
 DO
     CP  R10, #(MAX_POSS_L_BLKS/16)
     IF  EQ  THEN
         CPB  IS_COMPACTED, #FALSE
         IF  EQ  THEN
             CALL COMPACT_L
             LD  R10, #ZERO
             LDB  IS_COMPACTED, #TRUE
         ELSE
             LD  R0, #LOCAL_MEMORY_FULL
             RET
         FI
     FI
     LD  R11, #ZERO
     LD  R12, LOCAL_MEM_BIT_MAP(R10)
     DO
         BIT  R12, R11
         IF  Z  THEN
             DEC  R0, #1
         ELSE
             LD  R0, BLKS
         FI
         CP  R0, #ZERO
         IF  EQ   THEN
             LD  R1, R10
             MULT  RR0, #16
             ADD  R1, R11
             SUB  R1, BLKS
             MULT  RR0, #BLK_SIZE
             ADD  R1, #LOCAL_MEM_BASE
             LD  R0, #VALID
             LD  R13, BLKS
             DO
                 LD  R12, LOCAL_MEM_BIT_MAP(R10)
                 DO
                     SET  R12, R11
                     DEC  R13, #1
                     DEC  R11, #1
                     CP  R13, #ZERO
                     IF  EQ  THEN
                         LD  LOCAL_MEM_BIT_MAP(R10), R12
                         RET
                     FI
```

```
                        CP  R11, #ZERO
                        IF  EQ  THEN
                            LD  LOCAL_MEM_BIT_MAP(R10), R12
                            LD  R11, #15
                            DEC  R10, #1
                            EXIT
                        FI
                    OD
                OD
            FI
            INC  R11, #1
            CP  R11, #16
            IF  EQ  THEN
                LD  R11, #ZERO
                EXIT
            FI
        OD
        INC  R10, #1
    OD
END ALLOC_LOCAL_MEMORY


FREE_LOCAL_BIT_MAP  PROCEDURE
   !*****************************************!
   ! PASSED PARAMETERS                       !
   !   R0 = BASE_ADDR                        !
   !   R1 = BLKS                             !
   ! LOCAL VARIABLES                         !
   !   R10 = COUNTER FOR BIT RESET           !
   !   R11 = BIT_MAP INDEX                   !
   !   R12 = BIT_MAP WORD                    !
   !*****************************************!
    ENTRY
    CLR  R10
    LD  R11, R0
    SUB  R11, #LOCAL_MEM_BASE
    DIV  RR10, #BLK_SIZE*16
    DO
        LD  R12, LOCAL_MEM_BIT_MAP(R11)
        DO
            RES  R12, R10
            DEC  R1, #1
            CP  R1, #ZERO
            IF  LT  THEN
                LD  LOCAL_MEM_BIT_MAP(R11), R12
                RET
            FI
            INC  R10, #1
            CP  R10, #16
            IF  EQ  THEN
                LD  LOCAL_MEM_BIT_MAP(R11), R12
```

```
                LD  R10, #ZERO
                EXIT
            FI
        OD
        INC  R11, #1
    OD
END FREE_LOCAL_BIT_MAP


FREE_GLOBAL_BIT_MAP  PROCEDURE
   !*****************************************!
   !  PASSED PARAMETERS                      !
   !    R0 = BASE_ADDR                       !
   !    R1 = BLKS                            !
   !  LOCAL VARIABLES                        !
   !    R10 = COUNTER FOR BIT RESET          !
   !    R11 = BIT_MAP INDEX                  !
   !    R12 = BIT_MAP WORD                   !
   !*****************************************!
    ENTRY
    CLR  R10
    LD  R11, R0
    SUB  R11, #GLOBAL_MEM_BASE
    DIV  RR10, #BLK_SIZE*16
    DO
        LD  R12, GLOBAL_MEM_BIT_MAP(R11)
        DO
            RES  R12, R10
            DEC  R1, #1
            CP  R1, #ZERO
            IF  LT  THEN
                LD  GLOBAL_MEM_BIT_MAP(R11), R12
                RET
            FI
            INC  R10, #1
            CP  R10, #16
            IF  EQ  THEN
                LD  GLOBAL_MEM_BIT_MAP(R11), R12
                LD  R10, #ZERO
                EXIT
            FI
        OD
        INC  R11, #1
    OD
END FREE_GLOBAL_BIT_MAP
```

```
ALLOC_GLOBAL_MEMORY  PROCEDURE
   !*****************************************!
   ! PASSED PARAMETER                        !
   !   R0 = BLKS OF MEMORY                   !
   ! RETURNED PARAMETERS                     !
   !   R0 = SUCCESS_CODE                     !
   !   R1 = BASE_ADDR                        !
   ! LOCAL VARIABLES                         !
   !   R0 = BLKS                             !
   !   R10 = BIT_MAP INDEX                   !
   !   R11 = COUNTER FOR BIT                 !
   !   R12 = BIT_MAP WORD                    !
   !   R13 = WORKING REGISTER                !
   !*****************************************!

    LOCAL  BLKS  WORD
           IS_COMPACTED  BYTE
           FILLER3   BYTE
    ENTRY
    LD  BLKS, R0
    LDB  IS_COMPACTED, #FALSE
    LD  R10, #ZERO
    DO
        CP  R10, #(MAX_POSS_G_BLKS/16)
        IF  EQ  THEN
            CPB  IS_COMPACTED, #FALSE
            IF  EQ  THEN
                CALL COMPACT_G
                LD  R10, #ZERO
                LDB  IS_COMPACTED, #TRUE
            ELSE
                LD  R0, #GLOBAL_MEM_FULL
                RET
            FI
        FI
        LD  R11, #ZERO
        LD  R12, GLOBAL_MEM_BIT_MAP(R10)
        DO
            BIT  R12, R11
            IF  Z  THEN
                DEC  R0, #1
            ELSE
                LD  R0, BLKS
            FI
            CP  R0, #ZERO
            IF  EQ   THEN
                LD  R1, R10
                MULT  RR0, #16
                ADD  R1, R11
                SUB  R1, BLKS
                MULT  RR0, #BLK_SIZE
```

```
                    ADD  R1, #GLOBAL_MEM_BASE
                    LD   RØ, #VALID
                    LD   R13, BLKS
                    DO
                        LD   R12, GLOBAL_MEM_BIT_MAP(R1Ø)
                        DO
                            SET  R12, R11
                            DEC  R13, #1
                            DEC  R11, #1
                            CP   R13, #ZERO
                            IF   EQ  THEN
                                 LD GLOBAL_MEM_BIT_MAP(R1Ø), R12
                                 RET
                            FI
                            CP   R11, #ZERO
                            IF   EQ  THEN
                                 LD   GLOBAL_MEM_BIT_MAP(R1Ø), R12
                                 LD   R11, #15
                                 DEC  R1Ø, #1
                                 EXIT
                            FI
                        OD
                    OD
                FI
                INC  R11, #1
                CP   R11, #16
                IF   EQ  THEN
                    LD   R11, #ZERO
                    EXIT
                FI
            OD
            INC  R1Ø, #1
        OD
END ALLOC_GLOBAL_MEMORY


READ_PAGE  PROCEDURE
    !****************************************!
    ! PASSED PARAMETERS                      !
    !   RØ = BLK_NO                          !
    !   R1 = BASE_ADDR                       !
    ! RETURNED PARAMETER                     !
    !   RØ = SUCCESS_CODE                    !
    ! LOCAL VARIABLES                        !
    !   R1Ø = COUNTER FOR BLOCK MOVE         !
    !   R11 = SIMULATED DISK ADDRESS         !
    !****************************************!
     ENTRY
     LDL  RR1Ø, #BLK_SIZE
     MULT  RR1Ø, RØ
     ADD  R11, #DISK_MEM_BASE
```

```
        LD  R10, #MAX_PAGE_SIZE
        LDIR  @R1, @R11, R10
        LD  R0, #VALID
    END  READ_PAGE


WRITE_PAGE  PROCEDURE
    !*****************************************!
    ! PASSED PARAMETERS                       !
    !   R0 = BLK_NO                           !
    !   R1 = FROM_BASE_ADDR                   !
    ! RETURNED PARAMETR                       !
    !   R0 = SUCCESS_CODE                     !
    ! LOCAL VARIABLES                         !
    !   R10 = COUNTER FOR BLOCK MOVE          !
    !   R11 = SIMULATED DISK ADDRESS          !
    !*****************************************!
        ENTRY
        LDL  RR10, #BLK_SIZE
        MULT  RR10, R0
        ADD  R11, #DISK_MEM_BASE
        LD  R10, #MAX_PAGE_SIZE
        LDIR  @R11, @R1, R10
        LD  R0, #VALID
    END  WRITE_PAGE

READ_SEGMENT  PROCEDURE
    !*****************************************!
    ! PASSED PARAMETERS                       !
    !   R0 = PAGE_TABLE_LOC (BLK_#)           !
    !   R1 = MEMORY_ADDR                      !
    ! RETURNED PARAMETER                      !
    !   R0 = SUCCESS_CODE                     !
    ! LOCAL VARIABLES                         !
    !   R2 = INDEX FOR PAGE_TABLE_ARRAY       !
    !   R10 = COUNT FOR BLOCK MOVE            !
    !   R11 = DISK_BLK_# CONV TO MEM ADDR     !
    !   R13 = DISK ADDRESS                    !
    !*****************************************!
        ENTRY
        LDL  RR10, #BLK_SIZE
        MULT RR10, R0
        ADD  R11, #DISK_MEM_BASE
        LD  R2, #ZERO
        DO
        LD  R10, #MAX_PAGE_SIZE
            LD  R13, R11(R2)
            MULT  RR12, #BLK_SIZE
            ADD  R13, #DISK_MEM_BASE
            LDIR  @R1, @R13, R10
            INC  R2, #1
```

```
        CP  R2, #MAX_PAGE_SIZE
        IF  EQ  THEN
            EXIT
        FI
        LD  R0, R11(R2)
        CP R0, #ZERO
        IF  EQ  THEN
            EXIT
        FI
    OD
    LD  R0, #VALID
END READ_SEGMENT


WRITE_SEGMENT  PROCEDURE
    !*****************************************!
    ! PASSED PARAMETERS                       !
    !   R0 = PAGE_TABLE_LOC (BLK_#)           !
    !   R1 = MEMORY_ADDR                      !
    ! RETURNED PARAMETER                      !
    !   R0 = SUCCESS_CODE                     !
    ! LOCAL VARIABLES                         !
    !   R10 = PAGE_TABLE ARRAY INDEX          !
    !   R11 = DISK_BLK_NO CONV TO MEM ADDR    !
    !   R13 = DISK ADDR                       !
    !*****************************************!
    ENTRY
    LDL  RR10, #BLK_SIZE
    MULT RR10, R0
    ADD  R11, #DISK_MEM_BASE
    LD  R2, #ZERO
    DO
    LD  R10, #MAX_PAGE_SIZE
        LD  R13, R11(R2)
        MULT  RR12, #BLK_SIZE
        ADD  R13, #DISK_MEM_BASE
        LDIR  @R13, @R1, R10
        INC  R2, #1
        CP  R2, #MAX_PAGE_SIZE
        IF  EQ  THEN
            EXIT
        FI
        LD  R0, R11(R2)
        CP R0, #ZERO
        IF  EQ  THEN
            EXIT
        FI
    OD
    LD  R0, #VALID
END WRITE_SEGMENT
```

```
READ_DISK_BIT_MAP  PROCEDURE
    !*****************************************!
    ! RETURNED PARAMETERS                     !
    !   R0 = SUCCESS_CODE                     !
    ! LOCAL VARIABLES                         !
    !   R10 = DISK_BIT_MAP_BUFF ADDR          !
    !   R11 = COUNTER FOR BLK MOVE            !
    !   R13 = BIT MAP DISK ADDR               !
    !*****************************************!
     ENTRY
     LD  R10, #DISK_BIT_MAP
     LD  R13, #DISK_BIT_MAP_LOC
     CLR R12
     MULT  RR12, #BLK_SIZE
     ADD  R13, #DISK_MEM_BASE
     LD  R11, #(MAX_POSS_D_BLKS/16)
     LDIR  @R13, @R10, R11
     LD  R0, #VALID
END  READ_DISK_BIT_MAP

WRITE_DISK_BIT_MAP  PROCEDURE
    !*****************************************!
    ! RETURNED PARAMETER                      !
    !   R0 = SUCCESS_CODE                     !
    ! LOCAL VARIABLES                         !
    !   R10 = DISK_BIT_MAP_BUFF ADDR          !
    !   R11 = COUNTER FOR BIT MAP             !
    !   R13 = BIT MAP ADDRESS                 !
    !*****************************************!
     ENTRY
     LD  R10, #DISK_BIT_MAP
     LD  R13, #DISK_BIT_MAP_LOC
     CLR R12
     MULT  RR12, #BLK_SIZE
     ADD  R13, #DISK_MEM_BASE
     LD  R11, #(MAX_POSS_D_BLKS/16)
     LDIR  @R10, @R13, R11
     LD  R0, #VALID
END  WRITE_DISK_BIT_MAP

SEARCH_DISK_BIT_MAP  PROCEDURE
    !*****************************************!
    ! PASSED PARAMETER                        !
    !    R0 = START_SRCH_BLK_#                !
    ! RETURNED PARAMETERS                     !
    !    R0 = SUCCESS_CODE                    !
    !    R1 = FREE_BLK_#                      !
    ! LOCAL VARIABLES                         !
    !    R10 = BIT COUNTER                    !
    !    R11 = BIT MAP INDEX                  !
    !    R12 = BIT MAP WORD                   !
```

```
    !****************************************!
     ENTRY
     CLR  R10
     LD  R11, R0
     DIV  RR10, #16
  !  R10 = REM,  R11 = QUOT     !
     DO
         LD  R12, DISK_BIT_MAP(R11)
         DO
             BIT  R12, R10
             IF  Z  THEN
                 SET  R12, R10
                 LD  DISK_BIT_MAP(R11), R12
                 LD  R1, R11
                 MULT  RR0, #16
                 ADD  R1, R10
                 LD  R0, #VALID
                 RET
             FI
             INC  R10, #1
             CP  R10, #16
             IF  EQ  THEN
                 LD  R10, #ZERO
                 EXIT
             FI
         OD
         INC  R11, #1
         CP  R11, #(MAX_POSS_D_BLKS/16)
         IF  EQ  THEN
             LD  R0, #SEC_SCOR_FULL
             RET
         FI
     OD
     LD  R0, #VALID
END  SEARCH_DISK_BIT_MAP


CLEAR_DISK_BIT_MAP  PROCEDURE
    !****************************************!
    ! PASSED PARAMETER                       !
    !     R0 = BLK_NO TO CLEAR               !
    ! LOCAL VARIABLES                        !
    !     R10 = BIT COUNTER                  !
    !     R11 = BIT MAP INDEX                !
    !     R12 = BIT MAP WORD                 !
    !****************************************!
     ENTRY
     CLR  R10
     LD  R11, R0
     DIV  RR10, #16
   !  R10 = REM, R11 = QUOT  !
```

```
    LD  R12, DISK_BIT_MAP(R11)
    RES  R12, R10
    LD  DISK_BIT_MAP(R11), R12
END  CLEAR_DISK_BIT_MAP


MEMORY_MOVE  PROCEDURE
   !****************************************!
   ! PASSED PARAMETERS                      !
   !    R0 = TO_ADDR                        !
   !    R1 = FROM_ADDR                      !
   !    R2 = SIZE IN BYTES                  !
   !****************************************!
    ENTRY
    CLR  R12
    LD  R13, R2
    RR  R13, #1
    LD  R12, R0
    LDIRB  @R12, @R1, R13
END  MEMORY_MOVE


GET_UNIQ_ID  PROCEDURE
   !****************************************!
   ! RETURNED PARAMETERS                    !
   !    R0 = SUCCESS_CODE                   !
   !    R1 = UNIQUE_ID                      !
   ! NOTE: WILL BE STORED ON SEC STOR       !
   !****************************************!
    LOCAL  WORK_SPACE_BLK  ARRAY [MAX_PAGE_SIZE  WORD]
           UNIQ_ID  WORD
    ENTRY
    LD  R0,  #SYSTEM_DATA_LOC
    LD  R1,  #WORK_SPACE_BLK
    CALL READ_PAGE
    CP  R0, #VALID
    IF  NE  THEN
        RET
    FI
    LD  R10, #ZERO  ! UNIQ_ID INDEX !
    LD  R13, WORK_SPACE_BLK(R10)
    LD  UNIQ_ID, R13
    INC  R13, #1
    LD  WORK_SPACE_BLK(R10), R13
    LD  R0, #SYSTEM_DATA_LOC
    LD  R1, #WORK_SPACE_BLK
    CALL WRITE_PAGE
    LD  R1, UNIQ_ID
END  GET_UNIQ_ID
```

```
MAIN_LINE  PROCEDURE
   ENTRY
   CALL ALLOC_LOCAL_MEMORY
   CALL HBUG
END MAIN_LINE
END M_MGR_2
```

APPENDIX C - SWAP_IN PLZ/ASM CODE

```
MEM_MGR MODULE

!   * * * *   VERS. 1.0   * * * *  !


CONSTANT
            FALSE               := 0
            TRUE                := 1
            AVAILABLE           := 0  ! AST ENTRY AVAILABLE !
            ACTIVE              := 1  ! AST ENTRY ACTIVE !
            ZERO                := 0
            NULL                := %0000
            NULL_PAGE           := 0
            HBUG                := %A900
            MONITOR             := %059A

              !    SUCCESS CODES    !
            INVALID             := 0
            VALID               := 1
            FOUND               := 2
            NOT_FOUND           := 3
            SWAPPED_IN          := 4
            SWAPPED_OUT         := 5
            SEG_ACTIVATED       := 6
            SEG_DEACTIVATED     := 7
            SEG_CREATED         := 8
            SEG_DELETED         := 9
            LEAF_SEG_EXISTS     := 10
            NO_LEAF_EXISTS      := 11
            G_AST_FULL          := 12
            L_AST_FULL          := 13
            IN_LOCAL_MEMORY     := 14
            NOT_IN_LOCAL_MEM    := 15
            LOCAL_MEMORY_FULL   := 16
            GLOBAL_MEM_FULL     := 17
            VIRTUAL_CORE_FULL   := 18
            DUPLICATE_ENTRY     := 19
            NO_CHILD_TO_DEL     := 20
            SEC_STOR_FULL       := 21
            DISK_ERROR          := 22
            ALIAS_DOES_NOT_EXIST  := 23


      !    ATTRIBUTE MASKS    !
                READ_MASK         := %(2)11111110
                WRITE_MASK        := %(2)00000001
```

```
            CHANGED_MASK        := %(2)01000000
            IN_MEMORY_MASK      := %(2)00000100
            CLEARED             := 0                ! CLEAR ATTR !
    !    AUTHORIZED ACCESS      !
            READ                := 0
            WRITE               := 1
            EXECUTE             := %(2)00001000

    !   G_AST  FLAG BITS FIELD MASKS    !
            WRITABLE_MASK       := %(2)00000010
            WRITTEN_MASK        := %(2)00000100

    !   DESIGN  PARAMETERS      !
            BLK_SIZE            := 128
            NO_OF_PROCESSORS    := 1
            MAX_DBR_NO          := 4  ! EVEN NO. OF DBR_#'S !
            G_AST_LIMIT         := 16  ! MAX ENTRIES IN G_AST !
            L_AST_LIMIT         := 16  ! MAX ENTRIES IN L_AST !
            MAX_ENTRY_NO        := 21  ! SIZE OF ALIAS TABLE !
            NO_SEG_DESC_REG     := 8 ! NO. OF SEGMENT/PROCESS !
            FST_POSC_FREE_BLK:= 1

TYPE
            ADDRESS             WORD
            ALIAS_HEADER        RECORD [
                                  SEG_PAGE_TABLE_LOC  WORD
                                  PAR_ALIAS_TABLE_LOC WORD ]

         SEG_DESC_REG           RECORD [
                                  BASE_ADDR   ADDRESS
                                  LIMIT       BYTE
                                  ATTRIBUTE   BYTE ]

            ALIAS               RECORD [
                                  UNIQUE_ID       WORD
                                  CLASS           WORD
                                  SIZE            WORD
                                  PAGE_TABLE_LOC WORD
                                  ALIAS_TABLE_LOC WORD ]


            MMU                 RECORD [
                                  SDR ARRAY  [NO_SEG_DESC_REG
                                             SEG_DESC_REG]
                                  BLKS_USED       WORD
                                  MAX_BLKS        WORD]

            G_AST_REC           RECORD [
                                  UNIQUE_ID1      WORD
                                  GLOBAL_ADDR     ADDRESS
                          ! ONLY ONE PROCESSOR !
```

```
                                    PROCESSORS_L_ASTE_NO  WORD
                                ! WRITTEN_BIT_AND_WRITABLE_BIT !
                                    FLAG_BITS         WORD
                                    G_ASTE_NO_PAR     WORD
                                    NO_ACTIVE_IN_MEMORY  WORD
                                    NO_ACTIVE_DEPENDENTS  WORD
                                    PAGE_TABLE_LOC1   WORD
                                    SIZE1             WORD
                                    ALIAS_TABLE_LOC1  WORD
                                    SEQUENCER         WORD
                                    INSTANCE1         WORD
                                    INSTANCE2         WORD ]

            L_AST_REC             RECORD [
                                    MEMORY_ADDR     ADDRESS
                                    SEGMENT_NO_ACCESS_AUTH ARRAY
                                        [MAX_DBR_NO   BYTE] ]
            HANDLE                RECORD [
                                    UNIQUE_ID2    WORD
                                    H_INDEX       WORD ]


GLOBAL

  !$SECTION  G_DATA  !

            G_AST                 ARRAY [G_AST_LIMIT  G_AST_REC]
            G_AST_LOCK            BYTE
            DISK_BIT_MAP_LOCK     BYTE

  ! $SECTION L_DATA  !

            MMU_IMAGE           ARRAY [MAX_DBR_NO  MMU]
            L_AST                ARRAY [L_AST_LIMIT  L_AST_REC]
            ALIAS_TABLE          RECORD [ HEADER   ALIAS_HEADER
                                          ALIAS_ENTRY  ARRAY
                                          [MAX_ENTRY_NO  ALIAS] ]
            DISK_BIT_MAP_BUFF    ARRAY [6   BYTE]
            PAGE_TABLE_BUFFER    ARRAY [BLK_SIZE   BYTE]
EXTERNAL


ALLOC_LOCAL_MEMORY  PROCEDURE
    ENTRY
END ALLOC_LOCAL_MEMORY


READ_SEGMENT  PROCEDURE
    ENTRY
END READ_SEGMENT
```

```
FREE_LOCAL_BIT_MAP  PROCEDURE
    ENTRY
END FREE_LOCAL_BIT_MAP


ALLOC_GLOBAL_MEMORY  PROCEDURE
    ENTRY
END ALLOC_GLOBAL_MEMORY


MOVE_TO_GLOBAL  PROCEDURE
    ENTRY
END MOVE_TO_GLOBAL

SIGNAL_OTHER_MEMORY_MANAGERS  PROCEDURE
   ENTRY
END SIGNAL_OTHER_MEMORY_MANAGERS

INTERNAL

UPDATE_MMU_IMAGE   PROCEDURE
   !*****************************************!
   ! PASSED PARAMETERS                       !
   !   R0 = DBR_#                            !
   !   R1 = SEGMENT_#                        !
   !   R2 = ADDR                             !
   !   R3 = ACCESS                           !
   !   R4 = LIMIT                            !
   ! LOCAL VARIABLES                         !
   !   R10 = WORKING REGISTER                !
   !   R13 = WORKING REGISTER                !
   !*****************************************!
    ENTRY
    LD  R10, #MMU_IMAGE
    LD  R13, #SIZEOF MMU
    MULT  RR12, R0
    ADD  R10, R13
    LD  R13, #SIZEOF SEG_DESC_REG
    MULT  RR12, R1
    ADD  R10, R13
    LD  @R10, R2
    INC  R10, #2
    LDB  @R10, RL4
    INC  R10, #1
    LDB  RL4, @R10
    CPB  RL3, #EXECUTE
    IF  EQ  THEN
        ANDB  RL4, #%(2)11110111
    ELSE
        ANDB  RL4, #%(2)11111110
    FI
```

```
    ORB  RL4, RL3
    LDB  @R10, RL4
    RET
END UPDATE_MMU_IMAGE


UPDATE_L_AST_ACCESS   PROCEDURE
   !*****************************************!
   ! PASSED PARAMETERS                       !
   !   R0 = INDEX                            !
   !   R1 = ACCESS_AUTH                      !
   !   R2 = DBR_#                            !
   ! LOCAL VARIABLES                         !
   !   R5 = WORKING REGISTER                 !
   !   R7 = WORKING REGISTER                 !
   !*****************************************!
    ENTRY
    LD  R5, #L_AST
    LD  R7, #SIZEOF L_AST_REC
    MULT  RR6, R0
    ADD  R7, #2
    ADD  R7, R2
    ADD  R5, R7
    LDB  RL3, @R5
    CPB  RL1, #WRITE
    IF  EQ  THEN
        ORB  RL3, #%(2)10000000
        LDB  @R5, RL3
    ELSE
        ANDB  RL3, #%(2)01111111
        LDB   @R5, RL3
    FI
    RET
END UPDATE_L_AST_ACCESS

CHECK_LOCAL_MEMORY  PROCEDURE
   !*****************************************!
   !  PASSED PARAMETERS                      !
   !    R0 = INDEX                           !
   !  RETURNED PARAMETER                     !
   !    R0 = TEST                            !
   !  LOCAL VARIABLES                        !
   !    R2 = I                               !
   !    R3 = SEG_NO                          !
   !    RH3 = ATTRIBUTES                     !
   !    R10 = ADDR OF MMU_IMAGE.SDR[SEG#]    !
   !    R11 = ADDR OF L_AST[R0].SEG/ACC[I]   !
   !    R12,13 = WORKING REGISTERS           !
   !*****************************************!
   ENTRY
```

```
    LD  R2, #ZERO
    DO
        CP  R2, #MAX_DBR_NO
        IF  EQ  THEN
            LD  R0, #NOT_IN_LOCAL_MEM
            RET
        FI
        LD  R11, #L_AST
        LD  R13, #SIZEOF L_AST_REC
        MULT  RR12, R0
        ADD  R11, R13
        ADD  R11, #2  ! SEGMENT NO OFFSET !
        ADD  R11, R2
        LDB  RL3, @R11
        CLRB  RH3
        ANDB  RL3, %(2)01111111
        CPB  RL3, #ZERO
        IF  NE  THEN
           LD  R10, #MMU_IMAGE
           LD  R13, #SIZEOF MMU
           MULT  RR12, R2
           ADD  R10, R13
           ADD  R10, R3
           ADD  R10, #3    ! ATTRIBUTES OFFSET !
           LDB  RH1, @R10
           ANDB  RH1, #IN_MEMORY_MASK
           CPB  RH1, #ZERO
           IF  NE  THEN
               LD R0, #IN_LOCAL_MEMORY
               RET
           FI
        FI
        INC  R2, #1
    OD

END CHECK_LOCAL_MEMORY


CHECK_MAX_VIRTUAL_CORE  PROCEDURE
    !****************************************!
    !  PASSED PARAMETERS                     !
    !    R0 = DBR_#                          !
    !    R1 = BLKS                           !
    !  RETURNED PARAMETER                    !
    !    R0 = SUCCESS_CODE                   !
    !  LOCAL VARIABLES                       !
    !    R10,R12 = WORKING REGISTERS         !
    !****************************************!
     ENTRY
     LD  R10, #MMU_IMAGE
     LD  R13, #SIZEOF MMU
```

```
        MULT  RR12, R0
        ADD   R10, R13
        LD    R13, #SIZEOF SEG_DESC_REG
        MULT  RR12, #NO_SEG_DESC_REG
        ADD   R10, R13
        LD   R12, @R10
        ADD   R12, R1
        INC   R10, #2
        CP   R12, @R10
        IF   GT  THEN
            SUB   R12, R1
            LD   R0, #VIRTUAL_CORE_FULL
        ELSE
            LD   R0, #VALID
        FI
        DEC   R10, #2
        LD   @R10, R12
        RET
END CHECK_MAX_VIRTUAL_CORE

SWAP_IN  PROCEDURE
    !****************************************!
    ! PASSED PARAMETERS                      !
    !   R0 = INDEX                           !
    !   R1 = DBR_#                           !
    !   R2 = ACCESS                          !
    ! RETURNED PARAMETER                     !
    !   R0 = SUCCESS CODE                    !
    !****************************************!
        LOCAL        INDEX          WORD
                     DBR_NO         WORD
                     ACCESS         WORD
                     G_AST_BASE     ADDRESS
        ENTRY
        LD   INDEX, R0
        LD   DBR_NO, R1
        LD   ACCESS, R2
        LD   R5, #G_AST
        LD   R13, #SIZEOF G_AST_REC
        MULT RR12, R0
        ADD   R5, R13
        LD    G_AST_BASE, R5
        ADD   R5, #16      ! SIZE OFFSET !
        CLR   R6
        LD    R7, @R5
        DIV   RR6, #BLK_SIZE
        LD   R6, R7
        DEC   R5, #12      ! L_AST INDEX OFFSET !
        LD   R7, @R5
        LD   R0, R1
        LD   R1, R6
```

```
CALL  CHECK_MAX_VIRTUAL_CORE
CP  R0, #VIRTUAL_CORE_FULL
IF  EQ  THEN
    RET
FI
INC  R5, #4      ! NO_ACTIVE_IN_MEMORY OFFSET !
INC  @R5, #1
LD  R8, @R5
CP  ACCESS, #WRITE
IF  EQ  THEN
    DEC  R5, #4    !  OFFSET TO FLAG_BITS !
    LD  R4, @R5
    OR  R4, #WRITABLE_MASK
    LD  @R5, R4
FI
LD  R0, R7
CALL  CHECK_LOCAL_MEMORY
AND  R4, #WRITABLE_MASK
CP  R4, #0
IF  NE  THEN
    CP  R8, #1
    IF  GT  THEN
        CP  R0, #IN_LOCAL_MEMORY
        IF  NE  THEN
            LD  R0, R6
            CALL  ALLOC_LOCAL_MEMORY
            CP  R0, #LOCAL_MEMORY_FULL
            IF  EQ  THEN
                RET
            FI
            LD  R9, R1
            INC  R5, #8      ! PAGE_TABLE_LOC OFFSET !
            LD  R0, @R5
            CALL  READ_SEGMENT
            CP  R0, #VALID
            IF  NE  THEN
                LD  R0, R9
                LD  R1, R8
                CALL  FREE_LOCAL_BIT_MAP
                RET
            FI
            LD  R10, #L_AST
            LD  R13, #SIZEOF L_AST_REC
            MULT  RR12, R7
            ADD R10, R13 !MEMORY_ADDR OFFSET INTO L_AST!
            LD  @R10, R9
        ELSE
            LD  R10, #L_AST
            LD  R13, #SIZEOF L_AST_REC
            MULT  RR12, R7
            ADD  R10, R13
```

```
                LD  R9, @R10
            FI
        FI
    ELSE
        LD  R8, R0
        LD  R5, G_AST_BASE
        INC  R5, #2    ! GLOBAL_ADDR  OFFSET !
        LD  R12, @R5
        CP  R12, #NULL
        IF EQ THEN
           LD  R0, R6
           CALL  ALLOC_GLOBAL_MEMORY
           CP  R0, #GLOBAL_MEM_FULL
           IF  EQ  THEN
               RET
           FI
           LD  R9, R1
           CP  R8, #IN_LOCAL_MEMORY
           IF  EQ  THEN
               LD  R0, R7
               INC  R5, #14    ! SIZE  OFFSET !
               LD  R2, @R5
               CALL  MOVE_TO_GLOBAL
               CP  R0, #VALID
               IF  NE  THEN
                   RET
               FI
            ELSE
               LD  R0, R1
               LD  R1, INDEX
               CALL  SIGNAL_OTHER_MEMORY_MANAGERS
               CP  R0, #VALID
               IF  NE  THEN
                   RET
               FI
           FI
       ELSE
          LD  R5 ,G_AST_BASE
          ADD  R5, #2    ! GLOBAL_ADDR OFFSET !
          LD  R9, @R5
       FI
    FI
    LD  R0, DBR_NO
    LD  R10, #L_AST
    LD  R13, #SIZEOF L_AST_REC
    MULT  RR12, R7
    ADD  R10, R13
    ADD  R10, R0
    INC  R10, #2
    LDB  RL1, @R10
    LD  R2, R9
```

```
      LD  R3, ACCESS
      LD  R4, R6
      CALL  UPDATE_MMU_IMAGE
      LD  R0, R7
      LD  R1, ACCESS
      LD  R2, DBR_NO
      CALL  UPDATE_LAST_ACCESS
      LD  R0, #SWAPPED_IN
END SWAP_IN

MAIN_LINE  PROCEDURE
      ENTRY
      CALL SWAP_IN
      CALL HBUG
END MAIN_LINE
END MEM_MGR
```

LIST OF REFERENCES

1. O'Connel, J. S., and Richardson, L. D., Distributed Secure Design for a Multi-microprocessor Operating System, MS Thesis, Naval Postgraduate School, June 1979.

2. Parks, E. J., The Design of a Secure File Storage System, MS Thesis, Naval Postgraduate School, December, 1980.

3. Coleman, A. R., Security Kernel Design for a Microprocessor-Based, Multilevel, Archival Storage System, MS Thesis, Naval Postgraduate School, December 1979.

4. Reitz, S. L., The Implementation of the Security Kernel for a Multi-microprocessor Operating System, MS Thesis, Naval Postgraduate School, June 1980.

5. Schell, Lt.Col. R. R., "Security Kernels: A Methodical Design of System Security," USE Technical Papers (Spring Conference, 1979). pp 245-250, March 1979.

6. Organick, E. J., The Multics System: An Examination of Its Structure, MIT Press, 1972.

7. Millen, J. K., "Security Kernel Validation in Practice," Communications of the ACM, v. 19 no. 5 p. 243-250, May 1976.

8. Madnick, S. E., and Donovan, J.J., Operating Systems, McGraw Hill, 1974.

9. Denning, D.E., "A Lattice Model of Secure Information Flow," Communications of the ACM, v. 19 p. 236-242, May 1976.

10. Reed, P. D., and Kanoidia, R. K., "Synchronization With Eventcounts and Sequencers," Communications of the ACM, v. 22 no. 2 p. 115-124, February 1979.

11. Reed, P. D., Processor Multiplexing In a Layered Operating System, MS Thesis, Massachusetts Institute of Technology, MIT LCS/TR-167, 1979.

12. Zilog, Inc., Z8010 MMU Memory Management Unit, Preliminary Product Specification, October 1979.

13. Riggins, C., "When No Single Language Can Do the Job, Make It a Language-Family Matter," Electronics Design, February 15, 1979.

14. Schell, Lt.Col. R. R., "Computer Security: the Achilles Heel of the Electronic Air Force?," Air University Review, v. 30 no. 2 p. 16-33, January 1979.

15. Saltzer, J. H., Traffic Control in a Multiplexed Computer System, Ph.D.Thesis, Massachusetts Institute of Technology, 1966.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
4.	Lyle A. Cox, Jr., Code 52Cl Department of Computer Science Naval Postgraduate School Monterey, California 93940	4
5.	LTCOL Roger R. Schell, Code 52Sl Department of Computer Science Naval Postgraduate School Monterey, California 93940	5
6.	Joel Trimble, Code 221 Office of Naval Research 800 North Quincy Arlington, Virginia 22217	1
7.	LT Alan V. Gary 3320 W. Epler Ave. Indianapolis, Indiana 46217	2
8.	LCDR Edmund E. Moore NAVELEXSYSCOM PME 107 Washington, D.C. 20360	1
9.	CAPT John L. Ross 107 Readon St. Weatherford, Texas 76086	1
10.	LT Hal R. Powell 1295 Heatherstone Way Sunnyvale, California 94087	1

11. Office of Research Administriation
Code 012A
Naval Postgraduate School
Monterey, California 93940 — 1

12. Uno R. Kodres, Code 52Kr
Department of Computer Science
Naval Postgraduate School
Monterey, California 93940 — 1

13. I. Larry Avrunin, Code18
DTNSRDC
Bethesda, Maryland 20084 — 1

14. R. P. Crabb, Code 9134
Naval Oceans Systems Center
San Diego, California 92152 — 1

15. Kathryn Heninger, Code 7503
Naval Research Lab
Washington, D.C. 20375 — 1

16. Dr. J. McGraw
U.C. - L.L.L. (1-794)
P.O. Box 808
Livermore, California 94550 — 1

17 Mark Underwood
NPRDC
San Diego, California 92152 — 1

18. Walter P. Warner, Code K70
NSWC
Dahlgren, Virginia 22448 — 1

19. M. George Michael
U.C. - L.I.L. (I-76)
P.O. Box 808
Livermore, California 94550 — 1